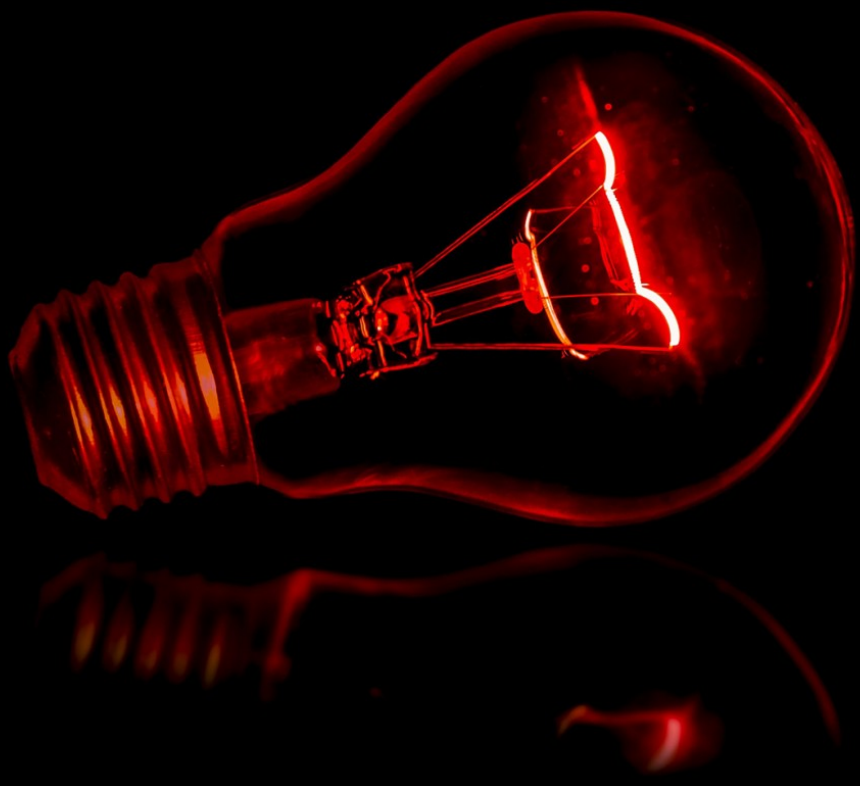
UMBRACO .NET CORE MASTERY

Build a kick ass website within Umbraco 10 or 9



JON D JONES

Umbraco .NET Core Mastery

Jon D Jones

Welcome

Welcome, to book number three, Umbraco .NET Core Mastery! Firstly, thank you for buying this book, I hope you will get a lot of value from it. The idea to write a book on how to build a CMS-powered website using .NET Core first crept into my mind way back in 2017. Ever since that day, I have been patiently waiting for one of the main CMS vendors to release a version of their CMS that supported this new framework. The wait finally ended in 2022 with the release of Umbraco version 9 (V9). It is now possible to create a kick-ass CMS website using .NET Core baby!

It is hard to believe that Microsoft released .NET Core 1 over 6 years ago back in 2016. When I first learnt about this new framework I was very keen to start using it on my next project. Having the option to build a .NET website on a MAC and being able to reduce my client's infrastructure costs seemed like a nice step forward for the .NET ecosphere. Sad as it sounds, I can even remember the exact time and place where I first learnt about .NET Core! I was attending a developer meet-up in some pokey pub off of Liverpool Street, in London (maybe a few pints deep). The speaker told us about this cool new Framework and all the cool new things we would soon be able to do.

At the time, naively, I assumed I would be building CMS projects using .NET Core within a few months. I even published a YouTube video in January 2018 saying as much. As you will probably now know, the general community adoption of .NET Core has been slower than most imagined. As so much software had been written using .NET Framework, for most teams, the choice wasn't' as simple as scrapping their current project and re-writing everything from the ground-up again.

The teams who were able to re-write their applications to support .NET Core bumped into an additional issue. A majority of software vendors, like Umbraco, had not released versions that supported .NET Core. At the time there were over 7000 NuGet packages with the majority still written using .NET Framework. This loss of functionality meant that a lot of the teams who had committed to upgrading to .NET Core found that the time and effort to rewrite everything from scratch was deemed too expensive. Many teams simply had two choices, wait for better adoption, or use another programming language, like Node and Javascript that were future-proofed.

As the .NET ecosphere was getting so fragmented, Microsoft decided to make everyone's lives easier by incorporating .NET Core into ASP.NET. As of November 10 2020, .NET Core was incorporated into ASP.NET 5. Incorporating .NET Core into ASP.NET was a clever move by Microsoft. As each version of .NET has an end-of-life support status, creating a major version bump within ASP.NET, helped to speed up adoption rapidly. With the Microsoft announcement, the majority of software vendors had their hands forced. They were forced into playing catch-up. If they did not upgrade, they faced the risk of their application running on an unsupported legacy stack. Being willing to bump into a major security flaw using an end-of-life version of a framework is an extremely risky move. This is why the majority of software vendors started to release .NET Core compatible software around the same time as the ASP.NET 5 release date.

The last official day of support for .NET 4 was April 26 2022 and on September 28th 2021, Umbraco released its first fully compatible version of the CMS that worked on ASP.NET 5 using .NET Core. The team made it with 6 months to spare before the official Microsoft support ran out for .NET 4. The release of Umbraco v9 meant being able to use .NET Core with a top-class CMS!

This book is being released less than a year after that release date. With so much time and effort put into Umbraco v9, you would have thought this book would be on v9 development. If you thought that you are wrong!

After Umbraco v9 shipped, the Umbraco development team did not lose any momentum. On June 16 2022, Umbraco v10 was released! Umbraco v10 is the first version of the CMS that works with ASP.NET 6. Microsoft claims that ASP.NET 6 is over 40% more performant compared to ASP.NET 5 and based on my personal experiences this feels to be true.

Umbraco v10 provides better cross-platform compared to v9. Umbraco 10 can be used with the SQLite database engine, which can be used on Windows, Linux, and Mac. Developers are now free to build a Umbraco-powered site on the infrastructure that makes them the happiest. The other final benefit is security. A number of third-party dependencies, like ImageSharp, Umbraco Examine, and AngularJS, have all been upgraded as well.

Having all this new software to play with was amazing, however, there was one annoying aspect about these releases for me personally, timing. During the 2020 lock-down, I wrote a book on Umbraco development, Umbraco Secrets Exposed. The focus of that book is on building a website using Umbraco powered by the .NET Framework. I was hoping my book on Umbraco v8 would be relevant for a few years, however, that is not how the IT sector rolls.

The focus of this book is an introduction to .NET Core development within Umbraco. This book will focus more on the CMS. The book will cover everything from getting set up, building a page, what core APIs are available and what they offer, what all the screens do, and much more. This book has taken me 7 months and over one hundred hours to write. This book is a com-

pletely new re-write. No areas have been lifted and shifted from the last book, so expect all new content!

What You Can Expect From This Book

The aim of this book is to provide you with a solid resource that will teach you how to build a kick-ass website whether it be Umbraco v9 or v10, however, this raises the question, which version should you use? If you are starting a new project, you should install v10. As I started writing this book long before v10 was released, all of the tips and recommendations contained within this book have all been tested on Umbraco v9, however, throughout this book I am assuming that you are using v10.

The main reason why you should favour v10 is due to Umbraco's move to a new release cadence. In case you were not aware, the Umbraco release cadence was changed to be more aligned with the Microsoft way of working on September 28th 2021.

Unless you are really geeky, I am assuming you are less familiar with the Microsoft ASP.NET release process. A release cadence encompasses several principles like semantic versioning, Long-term Support (LTS) and End-of-Life (EOL). If you are really interested, you can learn more about Microsoft's support policy here). Fancy-sounding works aside, how does this affect you?

When releasing updates to the CMS, Umbraco like most companies follows the semantic versioning specification, semver.org. The semver specification defines how software should be versioned. The rules of SemVar state that when a vendor is releasing a new version of their software, if that version contains a breaking code change it should be released as a new major version of that software. If you look at the history of Umbraco releases in terms of breaking

changes, you will understand why we have versions v8, v9 and v10. As the CMS introduced a lot of breaking changes when it moved to .NET Core we had a major version bump and v9 was created. In order to support ASP.NET 6, Umbraco needed to introduce several breaking code changes, this is why we have a v10 release rather than a v 9.1!

The semantic versioning part of the Umbraco release cadence is not new. The main change in the cadence is around support. As Umbraco becomes a more grown-up enterprise-sized organization, the development and support team can only support so much. Trying to support old legacy versions of software is not practical. This is why there will be a new level of support associated with major version releases of the CMS.

When Umbraco releases a new major version of its software, some versions will be shipped with long-term support (LTS) while others will not. Versions that are released with long-term support by Umbraco will be supported and patched for roughly two years longer than versions that are shipped without it. Understanding if a version has LTS or not, can help companies to plan exactly when they want to upgrade.

Typically, when companies move to this type of release cadence it is because they are aiming to release (and potentially break) the software more rapidly. When you start making new major version changes yearly, it is really hard for clients and their software teams to know when they should invest the time upgrading. Without a release cadence, teams could be constantly upgrading to try and keep up. Without foresight, trying to keep up can be frustrating and can lead to teams picking other software (like Saas) where the maintenance overhead is lower.

When a version of Umbraco is classed as actively being supported, the Umbraco team pledge to provide updates and fixes to that version until its end of life. As of

January 2022, the level of support Umbraco provides has been split into two stages:

- Support stage: When a version is in the support stage, Umbraco will actively fix bugs, issues, regressions, incompatibilities, and security breaches. During the support stage, expect to see lots of minor versions of the software being released. The duration of the support period is based on when the software was shipped. The support period will last for 9 months after the initial release date for normal releases and 24 months for LTS releases.
- Security stage: After the support period runs out, Umbraco will still support the software, however, the amount of support reduces. The security stage starts on the same day as the support period expires. In the security stage, Umbraco will only ship release patches for security issues. For normal releases, the security phase lasts for 3 months after the support stage finishes. This period is extended to 12 months for LTS releases.

The takeaway from this new cadence is that some versions of Umbraco will be supported for a lot longer than others. There is obviously an investment of time and effort in upgrading. In the future, if you are considering upgrading your Umbraco instance, you should also consider if a version has long-term support, or not.

Umbraco did not release v9 with long-term support. At the time of me writing this, v9 support period has nearly run out, even though it is less than a year since it was first released! This is why if you are starting a new project now, pick v10!

To ensure that you fully understand the consequences of this new process, below shows how this new cadence maps to different versions of Umbraco. One thing to note about this list is that there is a slight

oddity around the supported dates in versions 7 and 8. As this cadence is new, Umbraco is giving existing customers some extra time to upgrade.

If Umbraco were to strictly follow the rules of the new cadence, versions 7 and 8 would be instantly classed as end-of-life. As Umbraco 7 and 8 have always been considered versions with long-term support, turning off support for both versions overnight would not be fair. To ensure that existing customers have enough time to upgrade, Umbraco considers the last minor releases for each release as each version's initial release date. IN essence, Umbraco is giving their customers 2 years to plan and migrate their sites off of either version!

Umbraco 9: Umbraco 9 was released on September 28 2022. As version 9 was not classified as long-term supported, the support period Umbraco offers on this release is a lot shorter. On non-LTS releases, Umbraco only offers a 9-month support period (which will occur on September 16 2022). On that date, the additional security phase starts. This security stage support period only lasts for 3 months. This means the end-of-life date for v9 will be December 16 2022.

Umbraco 10: Version 10 was released with long-term support. Released in June 2022, v10s support will run out in June 2025

Umbraco 11, 12, 13 and 17: Umbraco 11 and 12 will not have long terms support so the support will end a year after each version is released.

Umbraco 13 will be the next version that is released with long-term support. Planned for release in 2023, this means it will be supported until 2026.

After v13, the next version that will ship with long-term support will be Umbraco 17 which is planned for release in 2025.

More information about this process can be found from the Long-term Support (LTS) and End-of-Life

(EOL) page.

Now we know which version of the software to use, we can focus on building a kick-ass Umbraco project using ASP.NET CORE. The advice within this book is relevant to all major versions of Umbraco from v9 upwards. To prevent this book from being very clunky to read I need to clarify something. To prevent you from reading sentences with lots of caveats and duplication, I will mainly be using the term .NET Core instead of specifically writing .NET Core/.NET 5/.NET 6 everywhere. Additionally, instead of writing v9/v10 everywhere, I will default to using v9.

Just like trying to decide when to write he/she/it/non-binary/0/1 nowadays, if I reference either a version of Umbraco or a version of .NET Core do not take the exact version too literally. Assume that I am referring to the version you are interested in.

By reading this book you will learn the art of what is possible when building a website using Umbraco with .NET Core. There are a lot of fundamental changes when jumping between Framework and Core. Gone are things like the beloved `global.ascx` and `web.config` . Gone are a lot of the Umbraco-specific configuration files that used to be found within the `Config` folder, it's all change!

For many developers, this change from the familiar will feel a bit daunting, however, do not fret. In your hands, you are holding a resource that will teach you everything you need in order to thrive in the world of Umbraco. Within this book, I aim to help improve your Umbraco architecture, design, development and deployment knowledge.

One of the great things about Umbraco compared to some of the new headless CMS solutions out there is that it ships with a lot of capabilities. Within Umbraco you can fully customise the backend, you can create a secure member area and even manage the members within the CMS. Member management may sound like

an obvious capability that every CMS can handle, however, in a lot of the newer best-of-breed SAAS CMS systems, member management is not included. In pretty much all of the newer CMS systems, if you bump into a membership requirement, you will need to pay for additional service (like Auth0). This is not the case with Umbraco!

I think this ability to customise the CMS to work how you need it to, combined with its vast array of capabilities is one of the main reasons why Umbraco is so great. As Umbraco is open-source, you can even fork the CMS on GitHub if you really want to!

If you pick Umbraco to power a client's website, you will be able to build a website for a client and fit it to their specifications exactly. You will not need to compromise on your design and you will not get forced into buying additional services. You will have all the tools you need out of the box to build a killer website.

One area that I am not going to cover in this book is headless CMS development with Umbraco. It is possible to build a headless website using Umbraco Heartcore. Heartcore is a stripped-down SaaS version of Umbraco that comes with a GraphQL layer and a Javascript SDK.

As of writing, the headless features offer as part of the Heartcore service are not available for normal Umbraco development. If you want to go headless, your options are to use the Heartcore service, pay Umbraco to host the website for you and to accept that you will loose a lot of your power to customise the CMS fully, or to look elsewhere.

The headless story with Umbraco is not all doom and gloom. Umbraco's plans for its headless offering over the next few years look very promising. The current aim is that from v13 Umbraco will offer headless support with an API layer and a webhook for JAMStack out of the box. Until the Umbraco headless capability catches up, if a client were to ask me to build a head-

less website using a CMS, I must admit I would pick other CMS systems over Umbraco. I'm not saying this to put you off using Umbraco and headless. I am only saying that you should take caution and do your due diligence about what solution is correct for you. When Umbraco does go headless, I may even write a new book on the topic!

My Thanks To You

Before we dive in and start learning about Umbraco, again I would like to personally thank you for purchasing this book. I spend more time than people would imagine writing free content on my website just to help developers build websites in a less painful way.

I have gotten up at 5 am to create content (much harder in the winter), missed family time to create content on Saturdays and Sundays, I have even gotten in trouble with my wife for taking time out of my holidays and bank holidays to write. I make no money from 90% of the content I create. Some people tend to think I make a fortune from my website, this is not the case. I keep creating content every week, as I enjoy it and people seem to find value in it. It is only through my books that I recoup some of the money which goes into the hosting costs etc.. of my site. By purchasing a copy of this book, you are helping me to continue to produce content.

It takes a lot of effort to write a book. My process is normally waking up at 5 am and writing in a cold dark room for a few hours before being woken up by a baby. To be transparent, I will not make a lot of money from this book. I get sales most weeks from my books, however, the effort to write/reward ratio is never going to make me wealthy from writing. I mainly do it to help people build better things.

Fundamentally, the reason I still love development and continue to look at new and improved ways of writing

code is my love of craftsmanship. In order to be the best developer you can be and to build the best website that you are capable of, you need to continually up your game and learn new skills. Nowadays finding content on any topic is much easier compared to when I started writing blogs on my website in 2011. Finding good in-depth content on CMS systems is still limited, so with this book, I hope to fill that gap. With all that mushy stuff over and done with, let us build some cool shit!

Further Reading

- Umbraco Secrets Revealed
- Versioning and release cadence
- Umbraco release cadence
- Long-term Support (LTS) and End-of-Life (EOL)
- .NET and .NET Core Support Policy

Installing And Configuring Umbraco For Development

The first step when building a website using Umbraco is to install the CMS on your development environment and get it working. There are two different installation workflows you can use. As always, it is still possible to install and manage Umbraco from within Visual Studio using NuGet. As of Umbraco v9, it is also possible to install the CMS via the command-line using the .NET CLI. In this chapter, you will learn how to install Umbraco using both approaches. As we are working within an ASP.NET 5 solution, the project structure and the type of files that you will be working with will differ compared to an ASP.NET Framework project. After learning how to install the CMS, you will learn about all these differences.

If this is your first journey moving from ASP.NET Framework to ASP.NET Core, the .NET CLI tool might be new to you. The official blurb around the .NET CLI is that it is a new cross-platform tool for creating, restoring packages, building, running and publishing .NET applications. The application installation part of the CLI tool is pretty powerful. The CLI tool allows third-party vendors to build custom project templates. Developers can then install these custom templates and use them to easily create ASP.NET Core powered websites. As of Umbraco v9, there is a Umbraco based template that you can use. As you will shortly see, you can use this Umbraco based template to quickly and easily get a Umbraco powered project up and running with a few simple commands.

Using the CLI is easy, open up a terminal and type `dotnet`. You can view all the templates that are installed within your local environment using this command:

```
dotnet new --list -l
```

An example of this output is shown below:

```
Templates            Short Name Language    Tags
-------------------- ---------- ----------- -------------
Console Application  console    [C#],F#,VB  Common/Consol
Class library        classlib   [C#],F#,VB  Common/Librar
```

To create an application using a template, you use the `dotnet new` command. To install a specific template you pass the templates “Short Name” as an installation option using this command:

```
dotnet new -t web
```

This template powered approach will also allow you to define the exact version of the CMS that you want to install. When you install Umbraco, you will likely want to install a specific version. You can do this by specifying the version at the end of the command like this:

```
dotnet install web::9.0
```

Alternatively, you can use an asterisk rather than specifying a specific version. The asterisk denotes that the installer should use "any version":

```
dotnet install web::*
```

Setting Up Your PC For Success

Before you can get Umbraco up and running, you need to ensure that your PC has all the recommended pre-flight checks covered. If you do not have the correct software and patches installed, the installer will fail.

Trust me when I say that debugging a failed installation can waste a lot of your time. In order to have a stress-free day, do not skip this step. To make sure the installation is efficient, make sure you install all of these things:

- ASP.NET 5: This is needed to power the framework
- .NET CLI: This will be installed while you install ASP.NET 5
- Windows Hosting Bundle: This is needed to allow you to host your website in IIS
- Microsoft Visual Studio 2019 version 16.8 or higher: Visual Studio must be patched otherwise the installation process will fail
- IIS
- SQL Server 2019 with SSMS
- Docker (optimal)

Database Setup

The job of a CMS is to store content and settings related to your website. This means that Umbraco needs its own database to store all that data. As part of the installation process, you will need to decide how this database should be hosted.

If you run through the installation process using the default options, the installer will install the database as a file-based database. The installer will create the database within your website's webroot. Two files will be created. One file will have an extension-type of `.mdf`, likely called `Umbraco.mdf`. This is the file that will contain all of your sites configuration and settings. The second file will have a file extension of `.ldf`, likely called `Umbraco_log.ldf`. This is an associated logging database that will be used by SQL.

Within Umbraco 8 and below these database files would be installed within a folder called `App_Data`. In Umbraco 9, these files can now be found in `Umbraco\Data`. We will come back to the `umbraco` folder shortly, however, as a general rule, if you need to find some CMS specific file, look within the `umbraco` folder first.

In order to get a sample site up and running quickly, a file-based SQL database will suffice, however, working this way is not optimal for a develop-

ment set-up. If you want your Umbraco powered website to run optimally, you will be better off hosting the database using SQL server.

When your database is hosted within SQL, you can use SQL Server Management Studio (SSMS) to work with the database more efficiently. Using SSMS means that you can easily make back-ups, restore your database to an earlier state, run SQL queries against the database and even publish updates to remote sources (like Azure) easily. SQL Server also comes with a free edition, so you do not need to pay for it.

Installing SQL Server on your development environment is simple enough. To get started you will need to follow these steps:

- **Install SQL Server Express**: You can download SQL Server Express from here. I recommend using the version that comes with SQL Server Management Studio. Be aware that the default download does not have it included. More information about be found here
- **Set SQL server to run in mixed-mode authentication**: In order for your website to communicate with a database hosted within a SQL server, you need to configure SQL to allow remote access. By default, this type of access is disabled. You will need to enabled `SQL and Windows Authentication` mode. The steps to do this are listed below:
- Open SQL Server Management Studio- Log in with an admin account using Windows Authentication - On `Object Explorer`, right-click on the server name and go to `Properties`
- Select the `Security` section
- In the `Server Authentication` tab change the selection from `Windows Authentication` mode to `SQL Server and Windows Authentication` mode
- Click `Ok`
- **Create/enable a SQL account**: To allow a website to communicate with a database it will need a connection string. The connection string is made up of four basic parts. The hostname or the IP of the server where the database lives, a username and a password that is allowed to access the database and the database name.

You will need to enable an account within SQL for remote access to work. You can either use an existing account or create a new account. This account will contain the username and the password you will need to add within your websites connection string will use.

By default, SQL will be installed with an account called `sa`. This is the admin account. This is the account I use in my development environment. I do not recommend that you ever use `sa` in production, however, in development it is fine. If you want to use the `sa` account, you need to make sure the account is enabled and has been configured for use for remote access. You can do this within SQL:

- In Object Explorer expand the `Security` tab
- Go to `Logins`
- Right click on the `sa` account and select `Properties`
- On the `General` tab set the password
- On the `Status` tab, in `Login` set it to `Enabled`
- Click `OK`
- **Create a blank database**: The Umbraco installer will install all the database tables, indexes and SPROCs that Umbraco needs to function. It will not create the database in SQL for you though. In order for the installer to function, you need to create a blank database first otherwise the installer will fail. You can name the database whatever you like, the main point is that it exists and your SQL account has enough permissions to access it!

With these four points taken care of, you are ready to install Umbraco.

How To Install Umbraco Using the CLI

From Umbraco v9 onwards, there is a new way to install Umbraco, via the CLI. Being able to install Umbraco solely within a terminal makes me happy. It also makes building out your CiCd pipeline that much easier.

To get going with Umbraco via the CLI, all you really need to know is three commands. In a terminal, make sure you navigate to the folder where you want to install the site. You can create a folder from the command line using this command:

```
mkdir foldername
```

The first Umbraco specific command that you will need to run is the command to install the Umbraco template. After installing this template, you can then use it to create as many blank Umbraco websites as you need. You can install the Umbraco template using this command:

```
dotnet new -i Umbraco.Templates
```

To confirm that the template was installed correctly, I recommend that you run the below command afterwards to check that a template named `umbraco` appears within the installed templates list:

```
dotnet new --list
```

With the Umbraco template installed, you can now create a new Umbraco project using this command:

```
dotnet new umbraco --name MyProject
```

Replace `MyProject` with your chosen project name. The installation process is quick and on average should take less than 2 minutes. After all the required files have been installed, you can run and start your new Umbraco website using this terminal command:

```
dotnet run
```

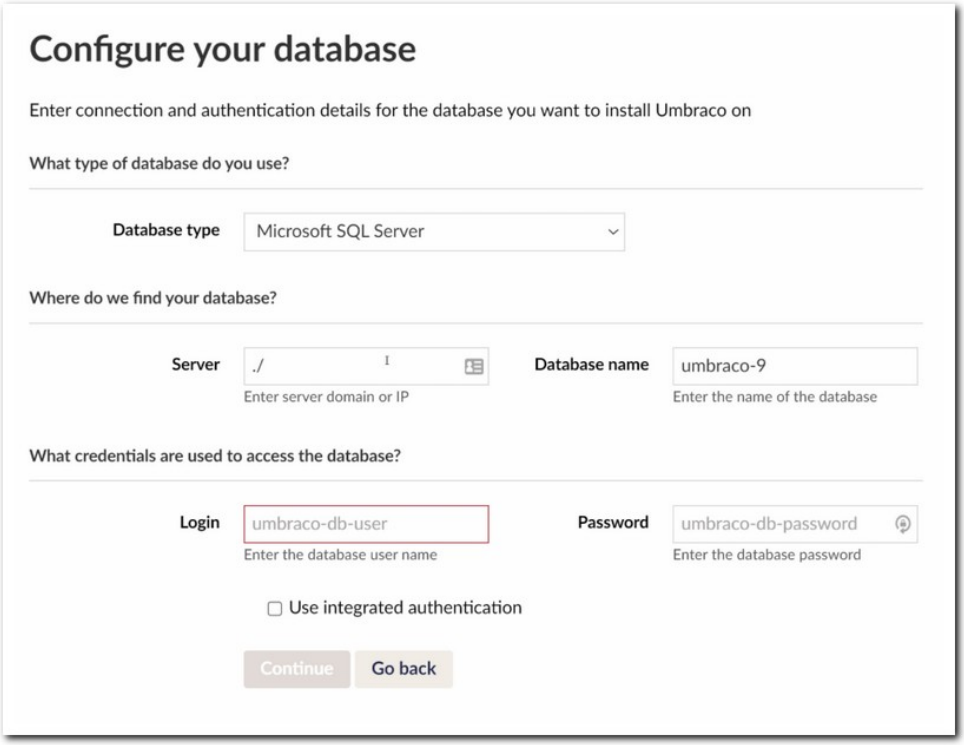
When the site starts, the URL that you will need to use to load the site will be displayed in the terminal window:

```
>dotnet run
Building...
[08:19:54 INF] Acquiring MainDom.
[08:19:54 INF] Acquired MainDom.
[08:19:56 INF] Now listening on: https://localhost:44353
[08:19:56 INF] Now listening on: http://localhost:57566
[08:19:56 INF] Application started. Press Ctrl+C to shut
[08:19:56 INF] Hosting environment: Development
[08:19:56 INF] Content root path: C:\Code\Umbraco9Defaul
```

I recommend that you use the `http` URL in order to access the CMS for the first time. Browsing to the hostname in a browser will launch the Umbraco installer. This installer is the thing that will install the database tables as well as configuring all the things.

The installer is comprised of a few main sections. On the first screen, you will need to prompt to create a Umbraco content editing admin account. You will use this account to access the CMS (also known as the backend). Do not forget these details, otherwise, you will not be able to log into the CMS!

On the next screen, you will need to configure the database hosting. If you followed the steps above, you should have SQL server ready to rock. Add the SQL server name/IP, the SQL username and password and the name of the blank database you created:



Configure SQL Server

After adding these details, select `Continue`. The installer will now go off and do its thang. If everything has gone according to plan, the Umbraco backend should load automatically within your browser.

If the backend does not load, you can try to launch it by adding `/umbraco` to the end of your development URL. After doing this, you should either get logged in to the CMS or, you will be presented with the login screen. To log into the CMS, you need to use the Umbraco admin details you entered during the Umbraco installation process:

At this point in time, the CMS will be empty. You have a blank canvas to start building your EPIC Umbraco powered website. A blank slate is great if you are already familiar with Umbraco. A blank slate is not so useful for developers who are new to Umbraco and unsure of where to start.

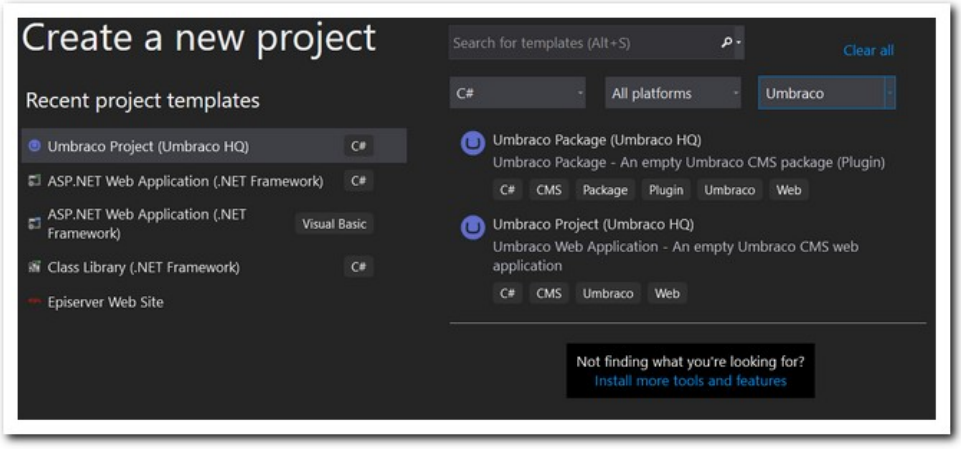
The first time you start using Umbraco, I recommend you also install one of the free community Starter Kits. As the name implies, a Starter Kit will install everything that you will require in order to get started within Umbraco v9. The Starter Kit will install a website within the CMS with all the document types and content to help you get familiar with how the CMS works. Installing a Starter Kit is dead simple and takes minutes. We will cover StartKits very shortly.

How To Install Umbraco Using Visual Studio

You can also install Umbraco 9 using the more traditional route, within Visual Studio. The only caveat is that the first step requires the use of a single CLI command. You will still need to install the Umbraco .NET template locally, which, at the time of writing can only be installed via the CLI. Installing this template is done using the command listed previously:

```
dotnet new -i Umbraco.Templates
```

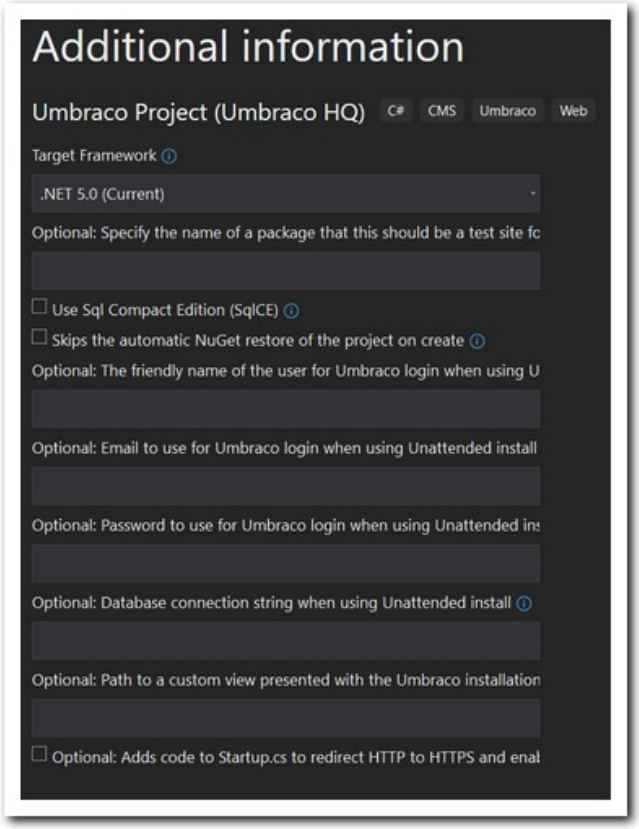
After you have the template installed successfully you can forget about the CLI tool and use Visual Studio. Fire up Visual Studio and create a new project. In the `All Languages` dropdown select `C#` and within the `All Project Types` dropdown select `Umbraco`:



Visual Studio Create New Project Screen

Select the `Umbraco Project` template and IGNORE the `Umbraco Package` template. Click `Next`, and enter in a project name and location.

I recommend avoiding using the word `Umbraco` in combination with the `.` character in your solution name. Having a name like `umbraco.mywebsite` can cause namespace clashes in your code when you try to call Umbraco APIs. Avoid making extra headaches for yourself by following a simple naming convention! When finished click `Next` to get to the final screen, `Additional Information`:



Visual Studio Additional Information Screen

It is possible to add all of your Umbraco project details to this screen, however, I recommend leaving all this stuff blank. You can fill in all of these settings very shortly within the normal Umbraco installation wizard!

With all the files required, all you need to do is click the `Debug` button within Visual Studio to launch the site. At this point the Umbraco installer will launch. You can then follow the steps listed above to configure the CMS!

Installing A Community Starter Kit

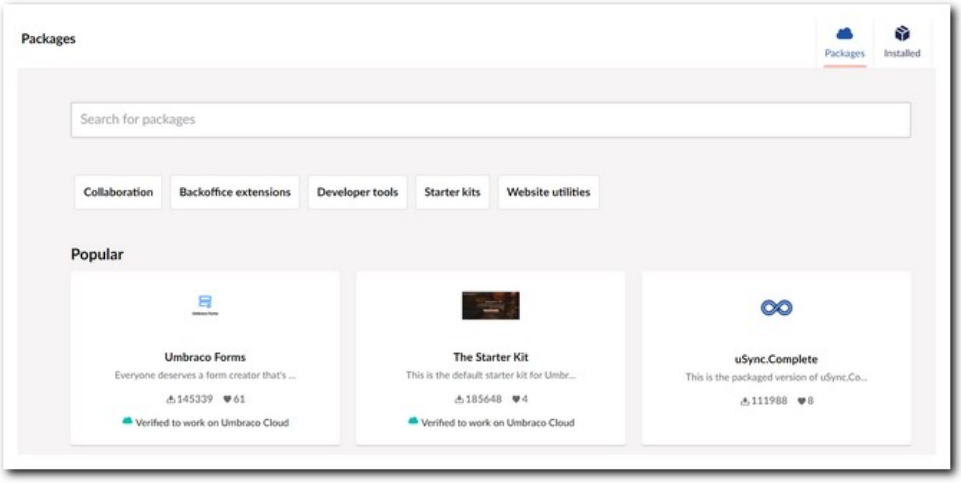
After the Umbraco installer has finished, you will be left with an empty website. During the initial CMS installation. The installer will not create any pages, add any content or images, or create any of the associated styling files. You will have a completely blank slate to build a new website.

You can prove this by trying to browse to the homepage. You will see nothing but a screen of beautiful white nothingness.

I know from first-hand experience, being faced with a blank slate can feel daunting the first time you undertake a project using a CMS you are unfamiliar with. When you are at the start of a project and you have nothing, it can be hard to know where to start! This is why I recommend that the next step after installing the CMS for Umbraco newbies is to install one of the free starter kits. I have always found that having a CMS populated with content, data-types, media and document types helps me to understand the internals of a CMS that much quicker.

In terms of creating the best possible site for a client, using a starter kit as the basis of your site architecture can have its downfalls. The starter kit might not be written to best practices and you might end up with extra bloat in your project that never gets used, however, this is where experience and the tips found within this book will help.

You can view most of the available starter kits for Umbraco v9 and upwards from within the Umbraco backend from the `Package` section. In previous versions of Umbraco, you used to be able to directly install packages from this screen. Due to the architectural differences between ASP.NET Framework and ASP.NET Core, the capability of installing packages from inside of the CMS is no longer possible. Packages now need to be installed via NuGet, which is not the end of the world. Just because you can not directly install packages from this screen anymore, the `Package` section is still a useful portal to peruse what packages you take advantage of. To view the available starter kits, open the Umbraco backend, go to the `packages` section and filter the results by `Starter Kits`:



Umbraco Package Browser

Umbraco's most popular starter kit is called [The Starter Kit]. Find `The Starter Kit` from the package search. From the package details page, you will get access to the commands to install the kit. You can install the starter kit via the CLI using this command:

```
dotnet add package Umbraco.TheStarterKit
```

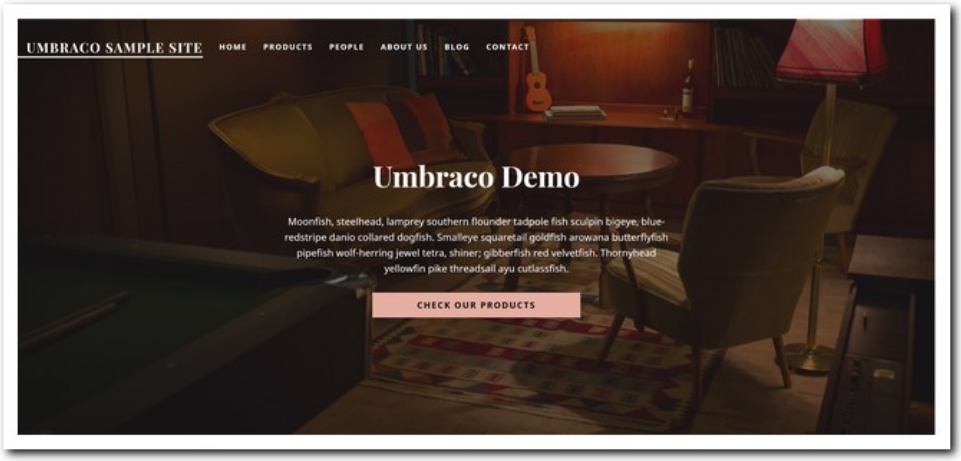
Alternatively, you can install the kit via NuGet. Within the Visual Studio package manager console, you can use this command:

```
Install-Package Umbraco.TheStarterKit -Version 10.0.0
```

After installing the package, rebuild the site by either doing a build within Visual Studio (`F5`) or via the terminal using:

```
dotnet run
```

When the site loads, you should see a working homepage that is populated with content and images:



Umbraco Starter Kit

You should not be in a much better position to start learning about the mighty power of Umbraco CMS.

Troubleshooting

I had issues getting the starter kit to work when I tried to install it with the CLI. If you encounter an error, check the logs to see what's gone wrong. The logs can be accessed from within the backend here:

`Settings` then `Log Viewer`

You can also get access to the actual log files from within the webroot, here:

```
umbraco\logs
```

If one approach fails, e.g. the CLI command I recommend trying the NuGet approach rather than trying to debug what hs gone wrong. This will save you a lot of time!

Hosting An Umbraco Website

In order to work efficiently while building a Umbraco project, you will have two needs. One will be to quickly test and validate code changes.

The other will be to perform tasks within the CMS, like creating new document types, creating pages, checking the logs, or, generating new models.

Using the debugger within Visual Studio to test your code changes is perfect. Unfortunately, the same can not be said when it comes to accessing the CMS. Having to open Visual Studio and wait for the site to launch whenever you need to make a change in the backend is not optimal workflow. It will result in you wasting a lot of time simply waiting for things to load. For these types of content tasks, life is much easier when you can access the CMS via a web server that provides persistent access.

Launching and accessing your site via the debugger is easy and can be done from within a terminal. Using the `dotnet run` command, your site will be launched using the new Kestrel server. Accessing the site this way is very handy while you are actively making development changes, however, not having a direct means of accessing the CMS will really slow your workflow down.

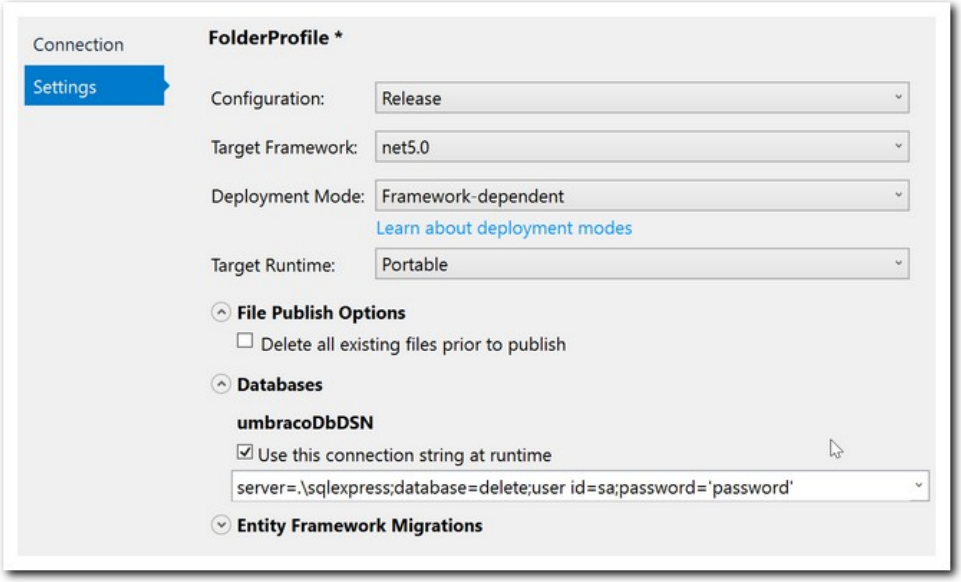
Historically, providing access to the CMS using ASP.NET Framework could be done very easily using IIS. You could point IIS to the files in your webroot. Making an update to the IIS hosted site was as simple as compiling the project. This unfortunately is no longer the case when we work within ASP.NET Core based projects. The mechanism of how the end website files are generated has changed. Before we get into the detail of hosting, it is important that you understand what this change is and how it affects you.

Publishing And Umbraco V9

In ASP.NET 5, you will need to run a publish command in order to generate the files required to run the site. Without performing a publish, the site will not be able to run. Performing a publish is simple enough. You can trigger a publish from the terminal using this command:

```
dotnet publish
```

You can alternatively trigger a publish by setting-up a publish profile within Visual Studio. Assuming your solution is open, click on the `Build > Publish` option from the top menu. The default publish profile settings will look similar to this:



Publish Profile Example

I do not recommend enabling the 'delete all files' option. If you do this locally, you will encounter locking issues. After publishing, within the console you should see the output folder location of the end files. On my laptop, my publish located could be found here:

```
C:\POC\umbraco9\bin\Debug\net5.0\publish\
```

The files that are generated from the publish command are the files that you will need to point your web server towards. If you inspect these files, you will notice that the files created by the publish command are very different compared to an ASP.NET Framework website. These changes are a result of the new ability to host an ASP.NET Core based website on either Windows, Linux or MAC.

Depending on where you want to host your website, will determine the flags that you will want to include during a publish. In ASP.NET 5, you can choose between two types of deployment modes, framework-dependent and self-contained:

Framework-Dependent Deployment: A framework-dependent publish will generate all the files that your solution requires to run without any framework code. In this mode, it is assumed that ASP.NET Core has been installed separately on the host machine. If the framework is not installed on the machine, your Umbraco website will fail to load.

The benefit of a framework-dependent publish, is that all the framework related code is omitted from the build process. This means that the size of the published package is smaller. If you need to host multiple websites on a single server, this type of publishing process can also help reduce overall memory usage on the server. Your websites can share the common framework components, reducing the number of things that need to simultaneously be loaded into active memory.

Self-Contained Deployment: A Self-Contained deployment is a new type of publishing option for .NET Core. In this mode, during the publishing

process, all the required platform-specific files are also generated and included within the output folder. As all the files that are required to run and host the website are included in the publish directory, the ASP.NET 5 framework does not need to be installed on the hosting machine. The side-effect of including all of these extra files is that the publish folder is a little larger compared to the output from a Framework-Dependent publish.

When you perform a self-contained deployment, you can target which type of output files will be generated. The output files for Windows, Mac or Linux will be slightly different. For example, if you want your site to run on a Windows machine you will get an `exe` to bootstrap your website. If you are targeting a MAC, you will have access to `dll` files that you can run by using the `dotnet MyWebsite.dll` command from a terminal.

If you're hosting multiple websites on a machine, a Self-Contained deployment can also increase reliability. As all the files required to run and host the site are created on an individual application basis, you can control the exact time that you upgrade each individual application. When you upgrade the framework on a hosted machine, in a framework-dependent set-up all the sites will be forced to use the new version at the same time.

Depending on which publish command you use will determine the files created within the output folder. The main output files that you need to be aware of include:

MyWebsite.exe`**: This is the starting file for your application. This is the file the server will call to lunch the site

web.config: An ASP.NET Core project will not contain a `web.config`, however, during the build process a very slimline `web.config` compared to ASP.NET Framework. After publishing a blank Umbraco sample site, your `web.config` will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet" arguments=".\JonDJones.dll" stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

Web.config example

When setting up your server, there is a very useful configuration setting you need to know about within this file `stdoutLogEnabled`. By default this will be disabled. If you struggle to get your web server working, changing this to `true` will start creating log files within the folder define in the `stdoutLogFile` key. If you encounter a random 500 error and you get stuck, enable this setting to uncover some useful debugging information

Your Website Binaries and Dependencies: All the code, Nuget packages and class libraries that you either create or reference will be copied

across.

Umbraco Binaries: All the files required to run the CMS will be bundled within the folder.

ASP.NET 5 or 6 Runtime: If you performed a self-containing publish, the ASP.NET 5/ASP.NET 6 runtime dependencies will be included.

File Locking Issue

Remembering to publish your site regularly on your local machine is a new chore for Umbraco v9/v10 developers, unfortunately, it is not the only one. There is are a unfortunate side-effects of switching to a publishing process. Here we will focus on the technical challenges. ASP.NET Core based application run as standalone applications. Trying to override files that are actively in-use during the publishing process will cause a problem, locked files.

On first publish, life is grand, however, if you try a subsequently publish to the site after it is being actively hosted, you will likely encounter a failed publish error. This error will be caused due to locked files.

An ASP.NET Core website is published as a self-executing application with a web-server. While this web-server is hosting the site, it will lock certain assemblies. Nightmare! As you would expect there are a few techniques to try and make this locking process less painful, which I will cover now:

Reset IIS: Simply turning off the website/server before publishing will release the lock. If you are hosting your website within IIS, you can use the `iisreset` command from within a terminal to reset IIS. Alternatively, you can also reset IIS via the UI . Open IIS and right-click on the top icon, click `stop`, publish the files and then click `start`.

Kill The Process: Killing the process that is locking the files will also release the lock. If you are presented with a message that contains the process Id (pid) like the one below:

> Visual studio reported this error message : The file is locked by: ".NET core Host (pid)"

You can use this terminal command to kill that process:

```
taskkill /f /pid 12345
```

Lock Hunter: If you use Windows, you can use a free third-party bit of software called Lock Hunter to unlock the files. After installing Lock Hunter, right-click on the published folder and select `What Is Locking This Folder?` from the Windows context-menu. If the folder is being locked, the program will list all the locking processes. Select all of them all then click on the `Unlock it!` button at the top. The publish process should now work.

.NET Core Host: If everything else fails, you can also try to kill the `.NET Core Host` service from within Task Manager.

The above solutions are all great to use on your local development server, what happens if you host your application on a server that you do not have access to? This is where `app_offline.htm` is handy.

App_Offline.html: Within an IIS hosted website, if you add a file called `app_offline.htm` into the root folder of a website, a magic thing will happen. IIS will automatically turn off your website and display the contents of the HTML file instead. When this happens, all locked files will be released. While this file exists in your website's root folder, your site will remain disabled. When the file gets removed, the site will come back online.

When you add `app_offline.htm` to the root folder, the server will throw a 503 error. You can copy `app_offline.htm` to the root folder manually, or, you can deploy it automatically via a script. One way of automating the file creation and deletion is during the publishing process using a MSBuild command. If you add the config found below at the bottom of your websites `csproj` file, whenever you trigger a publish, MSBuild will automatically create a `app_offline.htm` at the start of the process. When the file transfer part has completed it will then automatically delete the file:

```
<Project
  ToolsVersion="4.0"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
      <WebPublishMethod>MSDeploy</WebPublishMethod>
      <EnableMSDeployAppOffline>true</EnableMSDeployAppOffline>
    </PropertyGroup>
</Project>
```

Pre/Post Build Script

This script works using the `PreBuild` and `PostBuild` steps. `PreBuild` is triggered before publishing begins and `PostBuild` afterwards. Combining these two lifecycle methods will prevent your website from locking any current files during a publish!

MsDeploy: If you want to publish your website files remotely using an MSDeploy, you can also configure your build to do something similar. Within the publish profile, you can add a command that will create and delete `app_offline.htm` during the publishing process. You can create a publishing profile within Visual Studio, however, you can not apply this setting directly within the editor. Instead, you will need to manually edit the XML file. After you have a profile created, you can find your publishing profiles within the webroot in this location:

`Properties` then `PublishProfiles`

You will need to add an option called `EnableMSDeployAppOffline` and enable it:

Web Deploy

When `EnableMSDeployAppOffline` is enabled, this setting works in the same way as the one above. During the process, an `app_offline.htm` file will automatically be created and removed at the correct times.

Web Servers

Now we have a good understanding of the files that are generated during the publishing process and how we can publish those files without any locking issues, let us consider how to optimally build a website using Umbraco in our development environments. As mentioned at the start, I find the most efficient way to build a CMS project is to have a persistent way of accessing the CMS. To enable persistent access to Umbraco locally, you will need to host the site using a web server. When it comes to ASP.NET Core based applications, the good news is that there are some additional hosting options that were not previously available. When it comes to Umbraco v9/v10 hosting, you have three main ways to host your CMS locally:

- Host the website in Internet Information Server (IIS)
- Leave the Kestrel debugging server constantly running within an always open terminal window
- Set up a docker container with a Kestrel server

Traditionally, I have always used IIS to provide access to the CMS. Historically, there were a lot of good reasons for going this route. The main ones being that IS is the out-of-the-box web server for Windows and I could configure my IIS website by pointing the website at my webroot.

In the new world, my recommended approach for developers working with Umbraco is to use a container. As we have established, you need to perform a web publish to get the files required to run your website. In order to create a persistent IIS website for your Umbraco website to run within a container, you will need to constantly trigger a publish command whenever you make a code update. This will ensure the site is using the latest version of your code, however, it introduces one extra step into your workflow that wasn't previously required. Having to trigger a publish also introduces a synchronization issue around file management. Your

new environment can be configured to share the same database connection as your debugging website, so no CMS data will be lost. This is not the case for files.

If you uploaded some media, like an image, within the CMS, this media will not be uploaded into your webroot. Instead, it will be uploaded to a folder within the output folder of your publish command. The next time you fire up your local debugging server, these files will not be available within your webroot. These 'missing' files can break your website and cause it to look funky.

To fix these issues, you would need to manually sync any files created within the published website folder back to your webroot. This extra step not only wastes more of your precious time, but it can also be the cause of bugs. The main takeaway is that the old ways of working are no longer ideal. We need to mix it up.

As hosting your local persistent website in IIS is not as nice as it used to be, when it comes to v9 development, a better solution is to host your development environment in a docker container. Having your site hosted in docker will allow for persistent access to the CMS. Using a combination of two handy Docker features you can also eliminate the file sync issue. In Docker, you have commands to create volumes and mounts. A mount command can create a map from a folder within a container to a folder in the host machine. It is possible to configure your container so any files that are added within the CMS inside of the container are created directly within your webroot. Using a docker container will give you almost all the same benefits as the old workflow. The only caveat is building the docker file is a little bit more complex compared to setting up a site in IIS. You will still have to build the image when you make code changes, however, this can be scripted to save time.

There are other benefits from favouring the use of containers especially when it comes to production releases. Containers will allow you to perform like-live environment testing. If you use Kubernetes instead of Docker, you can also get the benefit of instant rollbacks. Your deployment process can be a simple matter of pointing the server to the container you want to make live. As long as you do not destroy the old container during the release, you will have a very quick rollback process. Specialists have written whole books around dev-ops and Kubernetes and the topic is too large to cover in justice here. If you want to learn more about setting up Kubernetes in production, Google is your friend. For us, we will focus on creating a Umbraco development environment using Docker.

Pre-requisites

To create a container, you will need to create a `Dockerfile`. Before we start creating this Docker file, you will need to ensure that your host machine has all the prerequisites ticked. Like most dev-ops tasks I've worked on, getting a Umbraco site to run in a container will take a lot of trial and error. If you try to get a container working, but, the host machine is not

configured correctly, you will waste hours of your life trying to figure out what has gone wrong. Setting this up can be frustrating, so do not skip this section for your own sanity!

Enable SQL Server Access: Umbraco uses a database to store all its data. It is possible to work with Umbraco using a file-based database, however, when using containers it's a better idea to host your database within SQL-server. To enable this architecture, your container will need to talk to the SQL instance installed on your host machine. Out-of-the-box, the type of SQL access the container will need is disabled. You will need to enable some things in SQL-Server before your container will run happily. Be warned, getting this access set-up can be fiddley. To make life as simple as possible, I recommend that you test this access first on your host machine, using SSMS. You can test access in SSMS, using your machine's local IP address and the local port your SQL instance is running under as the hostname. I will go over all these steps in this section.

To configure your SQL instance, you will need to use a tool called SQL Server Configuration Manager. This tool should be installed on your PC already, however, it is pretty well hidden. You can not find it from the Start menu. Instead, it is located within the `Windows` folder in one of two locations. Assuming you are using SQL 18, the SQL Configuration Manager executable can be found here:

```
C:\Windows\SysWOW64\SQLServerManager15.msc
```

Or here:

```
C:\Windows\System32\SQLServerManager15.msc
```

A list of which version of this tool you should use for your SQL instance can be found on the Microsoft website here. From within SQL Server Configuration Manager, you need to enable these things:

- Ensure TCP/IP for your SQL instance is enabled
- Ensure `Listen All` is enabled
- Get the IP and the Port number your instance is listening on
- Reset the server so the new changes are applied

If you are unsure of how to enable these options within SQL Server Configuration Manager, I have created a tutorial on my website that contains a video with the steps that you need to follow. You can find that video here. After applying these changes and resetting the server, the next step is to validate the updates have been successfully applied. You can do this by connecting to your database within SSMS using your machine's internal IP address and the SQL port number. I found the port number my SQL server was running under in SQL Server Configuration Manager within the `IP Address` tab in the `TCP Dynamic Ports` field. If you struggle to find this port, for reference the default SQL port is `1433`. When testing your connection in SQL Manager, use the username and password that you define in your connection string. To get your internal IP address, in a terminal run the `ipconfig` command. The internal IP will be

displayed against the `IPv4 Address` field and it should start with `192`. The format of the host string is IP and port, separated by a comma:

`192.168.0.25, 63636`

The end connection string will eventually look something like this:

```
"ConnectionStrings": {
    "umbracoDbDSN":
"Server=192.168.0.25, 63636;Database=umbraco9;user id=sa;password=password;"
},
```

Connection string

After you have successfully set up your container and got it talking to SQL, as a docker good practice you should be able to swap the IP and port within your connection string with this command `gateway.docker.internal`. Using `gateway.docker.internal` is a more robust approach to connection string management, however, for initial testing, focus on getting things working using the basics. Update the connection string to look like this:

```
"ConnectionStrings": {
    "umbracoDbDSN":
"Server=host.docker.internal;Database=umbraco9;user id=sa;password=password;"
}
```

Connection String Docker

The easiest way to use this connection string within your container is to create an environment within Visual Studio and create an environment-specific version of `appsettings.json`. It is likely that you will need to configure some other bits and bobs in this file in order to configure your site to work optimally within a container. This is why I think an environment-specific config file is a nicer approach compared to passing variables into the docker container on start-up.

If you use environment-specific config files, you will also need to make sure your website is configured to read in the correct environment settings file. The code to do this should be defined within your `Program.cs` file. An example of how the code you will need to add to make this work within `Program.cs` is shown below:

```
public static IHostBuilder CreateHostBuilder(string[] args)
{
  Host.CreateDefaultBuilder(args)
      .ConfigureLogging(x => x.ClearProviders())
      .ConfigureAppConfiguration((ctx, builder) =>
      {
          var environment = Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT");

          builder.AddJsonFile("appsettings.json", false, true);
          builder.AddJsonFile($"appsettings.{environment}.json", true, true);
          builder.AddEnvironmentVariables();
      })
      .ConfigureWebHostDefaults(webBuilder => webBuilder.UseStartup<Startup>());
}
```

Configuring AppSettings Environment

This lookup code is based on an environment variable called `ASPNETCORE_ENVIRONMENT`. If a corresponding `appsettings.{environment}.json` file exits it will apply those changes on top of the ones defined in `appsettings.json`:

Permissions: To set up a container for development success, you will need to create a file mount. You will need to map folders in your docker container to the corresponding folders in your webroot. This way you can upload things in the container and get access to them in your webroot. This will allow for the dual running of your site in docker and in a debugger. It will also mean you can easily add newly created files into source control.

In order to make this work, you will need to set the correct file permissions on the corresponding folders that you would like to set up mounts to on your host machine. If you forget to do this, your docker container will fail to run and you can waste hours of your life figuring out why the files are not being created how you expect them to… trust me. In your host machine, you need to ensure that all the folders you want to map have read/write access and are not read-only. For best practice, ensure the `docker-user` has read/write permission. As we are talking development environment and I am lazy, I simply tend to add the `Everyone` account with full permissions and replicate the permissions down the tree for simplicity.

Creating The Docker File

In order to host your website inside of a container, you will need to create a docker configuration file. I am going to use my Umbraco V9 starter kit as the base project for this example. You can clone this starter kit from my GitHub here.

The important thing to note with this site is that it is structured with two additional class libraries. I have added all the custom code used by the kit within one of the class libraries. The other library is used as the area

where the models are generated by Umbraco Model Builder. I will cover the Umbraco Model Builder in more detail later.

Whenever you are creating a new Docker image, you need to copy all the folders your website relies upon to the image. As I need to copy multiple folders, it makes sense for this project to create the docker file within the root solution folder. If you only have a single website project, you could create the docker file in your webroot instead if that floats your boat. Creating a docker file is easy, create a new blank file called `Dockerfile`:

```
echo > Dockerfile
```

Within the docker file, you will add the instructions of what is to be included within the image and the end container. The first thing you need to do is define the base container your image will use and then set the location where the files and folders will be copied to. This is done using the `FROM` and `WORKDIR` commands, like so:

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
```

The next instructions should tell Docker what files and folders it needs to copy into the image. You can do that by targeting all the `.csproj` files you want to copy and docker will do the rest. For each project, you should also call `dotnet restore` to ensure that each of the class libraries within the image has access to all the dependencies it requires to run. In terms of ordering, add the instruction to copy the main website project last:

```
COPY ["Umbraco.Models/Umbraco.Models.csproj",
      "Umbraco.Models/"]

RUN dotnet restore "Umbraco.Models/Umbraco.Models.csproj"

COPY ["Core/Core.csproj",
      "Core/"]

RUN dotnet restore "Core/Core.csproj"

COPY ["JWebsite/SampleSite.csproj",
      "Website/"]

RUN dotnet restore "Website/SampleSite.csproj"
```

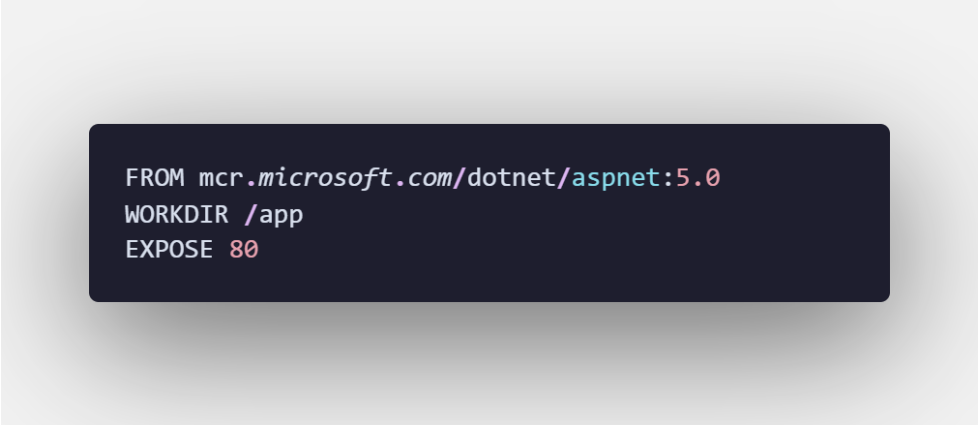
Docker Copy Command

The next step is to generate the website files that will be used in the image by performing a publish:

```
COPY . .
```

```
RUN dotnet publish "Website/SampleSite.csproj"
  -c Release -o /app/out
```

After you have all of the files required for the website to run within the image, the second part of the Dockerfile is to add the instructions on how a container should be instantiated. You need to specify the base image and the working directory again. This time the working directory is `app` and not `src`. Note, I am also exposing port `80`. I do this so we can access the website in the container via a web browser:



Set Container Environment

The main difference in this second section is the command to copy the output of the build to the publish folder:

```
COPY --from=build /app/out .
```

The final command is to start the container using `ENTRYPOINT` which takes two arguments. The first argument is the command Docker will run to start the ball rolling. Running an ASP.NET 5 site in a terminal is done using the `dotnet` CLI command. The second argument is the file the command should launch to start the site. In this example, the starter kits namespace is called `SampleSite`, so we kickstart the application using the `SampleSite.dll` assembly:

```
ENTRYPOINT ["dotnet", "SampleSite.dll"]
```

Build The Image

To create an image, you use the `docker build` command:

```
docker build -t umbraco9starterkit .
```

Using the `-t` flag, the command will create an image named `umbraco9starterkit`. You can then use this image to create a container. The `.` tells the command to look for the Dockerfile in the current directory. After you run this command and Docker has finished created the image, you can confirm it has been successfully created using the `docker images` command. You should see your new image returned in that list.

To create a container from an image, you use the `docker run` command. The `docker run` command that you will need to run to create and configure a container for a Umbraco website will be quite complex and will look something like this:

Basic Docker Run Command

Let us break down this command per flag so it is less overwhelming:

- `-rm`: Clean the container from any previous crap. You can omit this flag if you want to, however, it will mean your container will always start in a fresh state
- `-d`: Start the container as a background task. This command will prevent your terminal window from being locked up
- `-p`: Maps a port from your host machine to a port in the container. When you are trying to access a website in a container, it is best to not use port `80` on the host machine. If you are running legacy sites locally, you will still need to access IIS. In this example, I am using port `8080` to access the site. You can use any port you want as long as it is not already mapped to something else!
- `--name`: Specifies the container name. If you omit this flag docker will generate a random name for the container. Being able to easily identify the container, is very useful for debugging!
- `umbraco9starterkit`: The name of the image to create the container from
- `--env`: Pass any environment variables to the container

After running this command, your container should start and you should be able to access the website in a browser using `localhost` and port `8080`:

`http://localhost:8080`

Sometimes getting a container up and running is not straightforward. If you find yourself in this situation, you might want to inspect the file struc-

ture of your container. An easy way to check this structure is to use the `docker cp` command, like this:

```
docker cp umbraco:/app /app
```

This command will create a container that can be used to access your website. This command has not fixed the file synchronization issue. To resolve this issue, we need to update the container boot-up command to allow for file persistence.

Persistent Files

When a container is deleted, by default, any temporary files that have been created inside of it since it started will be deleted. It is possible to configure your container so these files are not lost. Within Docker, there are two commands that you can use to persist files. The strategy that you select, will be based on what you want to do with that data.

The recommendations shared in this section are focused on development. If you want to host your Umbraco website from a container in production, I recommend that you consider configuring Umbraco with a cloud-based storage provider, like Azure Blob Storage. The benefit of using a cloud storage provider is that all the files that are created within the CMS will be stored outside the container. This means file persistence will be handled for you by default. Multiple containers can be configured to use the same account so files can be accessed across environments. These files will also be externally backed-up. If you ever need to roll back your website to a previous state, a cloud-based provider can make life a lot easier. File providers will be covered in a later chapter.

When you are setting up a container to allow for an optimal development workflow, the focus is efficiency. This is where the docker `mount` capability is great for development. Folder mounts will allow you to map a folder in the host machine to a folder in the container. This means you can automatically access files in your solution folder when they are created within the container. Great!

Mounts are not the only way to deal with persistent data storage in Docker. Docker also has an additional file backup capability known as Volumes. Volumes are considered better practice compared to mounts, however, with a volume you can not map folders to the host machine.

If you research this topic online, you will quickly see that the general best-practice advice in this area recommends that you favour volumes over mounts. This makes sense in production. It is not very likely that you would want to map a folder within a container to a folder on the Kubernetes server.

Volumes will allow you to persist data and files when a container is destroyed. A Volume can be shared and accessed across multiple containers, so they are useful in load-balanced scenarios as well. The big difference between a mount is that a volume is a stand-alone area on disk, while a

mount maps to a folder in the host machine. You can learn more about docker volumes here.

Creating a volume folder in a development container is still extremely useful to persist certain types of files. For example, it makes more sense to map the `log` folder and the `temp` folder as volumes rather than mounts. Using volumes means that your temp data and logs will not get destroyed, improving performance, plus aiding you while you are debugging. For me, the big differentiator between the two strategies is source-control. If you want to add the created files into source-control use a mount. If you simply want to persist the data to speed up the container boot-up time use a volume.

To optimally boot up a Umbraco container, I recommend you use a mix of volumes and mounts. The big question is what will you need to map? Below lists the folders that I recommend you persist:

wwwroot: In ASP.NET Core based projects, the `wwwroot` is where all the public facing files live. When using the CMS, Umbraco will upload files to this folder. Whenever an asset or an image is added within the media library, it will be uploaded within this folder

Additionally, in the Umbraco editor, it's possible for a content editor to update and change the HTML, JS and CSS files that live within `wwwroot` . If you do not map `wwwroot` as a mount, these amends will be lost when the container is destroyed. As you will want to add these files into source control, the storage strategy on this folder makes sense to be set as a mount. The folder mapping for this rule will look like this:

- `website > wwwroot`

umbraco: Umbraco uses the `umbraco` folder to add CMS specific files that it needs. Within this folder, you can find the temp data folder, the frontend cache, search indexes as well as the site's log files. For performance, it can be handy to persistently store this data. The contents within this folder will not need to be added to source-control, so you can use a volume for this mapping rather than a `mount`. The folder mapping for this rule will look like this:

```
website > umbraco > data
```

```
website > umbraco > logs
```

usync: If you use the community Umbraco package uSync (covered later), you will want to store the output of a uSync export within source control. This means the file strategy should be a `mount`. The folder mapping for this rule will look like this:

```
website > uSync
```

Umbraco.Models: If you use the Umbraco Model Builder capability in `SourceCodeManual` mode, you will want to include the output directory in source-control. Models builder will be covered in more detail shortly. As we want to backup the output, this means the mapping will be a `mount`.

Taking all these mappings into account, this is how I structure the `docker run` command to also include all the mounts and volumes needed:

```
docker run -d -p 8080:80
   --mount type=bind,source="$(pwd)/Website/wwwroot",destination=/app/wwwroot
   --mount type=bind,source="$(pwd)/Website/uSync",destination=/app/uSync
   --mount type=bind,source="$(pwd)/Umbraco.Models",destination=/Umbraco.Models
   --mount type=bind,source="$(pwd)/Website/Views",destination=/app/Views
   --name umbraco
   umbraco9starterkit
   -v log-volume:/app/umbraco/Logs
   --env ASPNETCORE_ENVIRONMENT=Docker
```

Docker Run Command

Running this command will start your container with the correct mappings. With the container initialized with these settings, whenever a content editor updates files within the CMS, the files will be created within the host solution folder.

Host Umbraco Within IIS

The alternative way of hosting your Umbraco website locally is to use the IIS web server. As mentioned, for development I personally prefer using a container rather than IIS now. For completeness, I will also include the instructions on how to host a Umbraco site within IIS.

The good news is that the steps to set up a new website in IIS have not changed that much in ASP.NET Core based projects compared to ASP.NET Framework. The only big difference is how the worker process executes. In ASP.NET Core upwards, you need to configure the worker to run without ASP.NET, the server needs to trust that the website contains all the files it needs to run the site. Below outlines all the steps that you will need to follow in order to host a site successfully within IIS:

Windows Hosting Bundle: In order to run any ASP.NET Core and upwards website within a web server, you will need to install the files for the web server to work. Within .NET Core these hosting module files are called the Windows Hosting Bundle. You can download this module here here. Without this module, Umbraco will not load and you will encounter a funky error.

After installing this module, you will either need to reboot your PC, or restart IIS before the module can be used. You can reset IIS in the terminal using these two commands:

```
net stop was /y
net start w3svc
```

With this module install and IIS reset, you are ready to rock 'N roll!

Create A Website: The first step is to create a new website within IIS. You can do this by opening up IIS, and selecting `Create New Website`.

In order to configure this website to run ASP.NET Core code, you need to set the application pool to run in `No Managed Code` mode.

To do this, click on the `Application Pool` tag. Locate the application pool that is related to your website. Click on `Set Application Pool Defaults...` and set the property called `CLR version` to `No Managed Code`

Setup A Hostname: Installing a web-server (IIS) and setting up a website is easy, however, you will also need to configure a local hostname. This hostname is the URL that you will type into a browser to access the website. In order to set up a hostname, you will need to add an entry within your host file. The host file is located here:

```
C > Windows > System32 > etc > drivers
```

You will only be able to update the host file in a text editor that has administrator privileges. Within the host file, you need to create a map between your chosen hostname to localhost (127.0.0.1). Creating this map will mean that when someone types in the hostname in a browser, the request will be redirected to the local web server, instead of the internet. The hostname will be used to route the request to the correct website in IIS. To create this mapping is simple, add the IP address 127.0.0.1 first, add a tab and then add your chosen hostname, like this:

```
127.0.0.1 myWebsiteUrl
```

After we install Umbraco, when you type this hostname, `http://myWebsiteUrl` into a browser, your website should hopefully load. You will need to install Umbraco before this step will work!

You will know it has worked if you see an error (usually a 403 error)! Within IIS you will need to create a new website. In order to create a new website, you need three things, a name, a folder location and a hostname. Call the website in IIS anything you want as you will be the only person viewing it. Use the hostname you created above. Finally, point the folder location to the area on your PC where you will install the CMS files shortly.

Setup Permissions: When trying to view your website, it is also possible that the server will encounter Windows file access permission errors when trying to access the websites files. To fix the potential permission error, you need to set the correct folder permissions on your web folder. You can access the set permissions dialog from within IIS. Right-click on the website entry and select `permissions`:

To add a permission you will need to follow this process:

- Select the `Security` tab
- Click the Edit button

The security tab opens in a new window.

- Click `Add...`

The Select Users, Computers, or Groups dialog box appears.

- Select `Users`

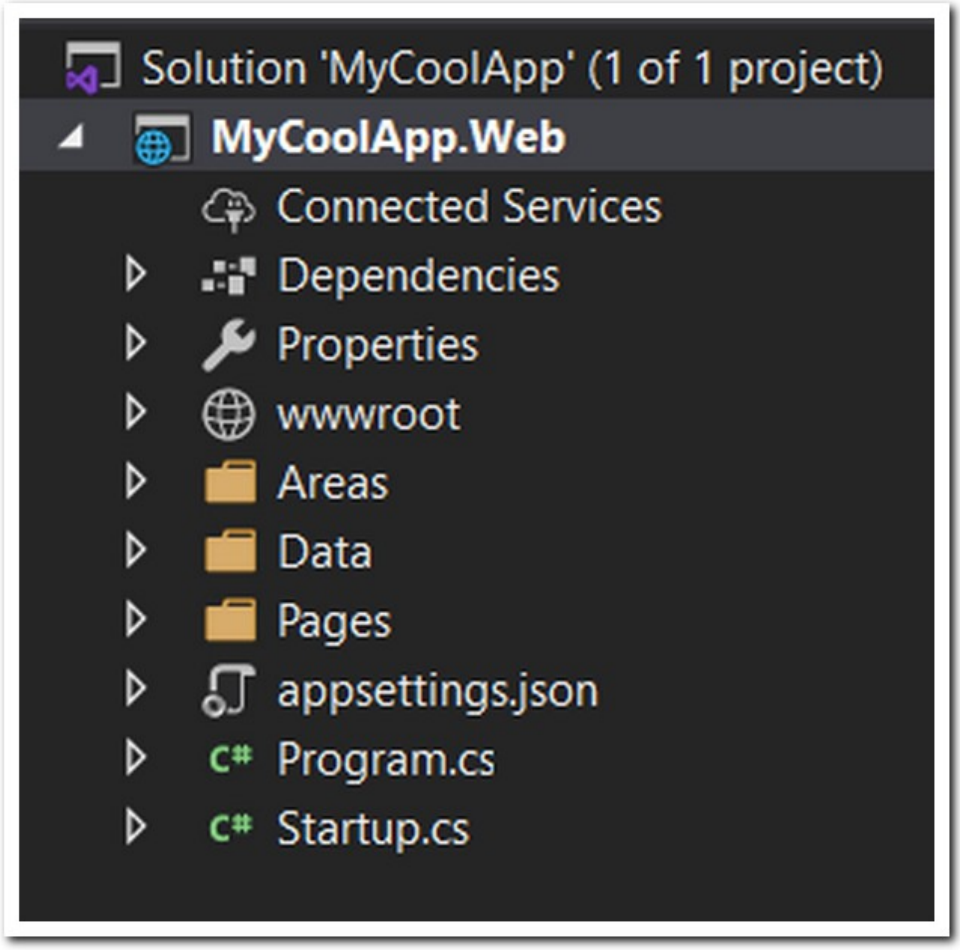
In the Enter the object names to select text box, type the name of the user or group that will have access to the folder. The permissions I recommend you add are:

- NETWORK SERVICE
- IIS AppPool\ApplicationPoolName
- Click OK

With all these steps covered, you should be able to connect to your website in a browser using your designated hostname.

Project Structure Explained

You should now have all the files required to run an Umbraco site and the ability to host that site in Docker or IIS. If you open the website project within your IDE of choice, you will notice that the files that the template and the installer generated look very different compared to previous versions of Umbraco.



Umbraco Starter Kit

The good news is that although at first glance it might look like you have a lot to learn, in reality, this new structure is more similar than different. The new project structure is fairly trivial to understand and is broken down into these components:

Connected Services: Connected services will be the first node in your project's structure and it will be the one you use the least. This section can be used to connect your project to an external service. To add a new connection, right-click on the node and select "Manage connected services". This will launch a wizard that will allow you to connect either to a dependency service, like a SQL database, Azure Storage or Application Insights. Alternatively, you can connect a Service Reference over OpenAPI, gRPC, or WCF. The wizard is useful for adding necessary service dependencies and giving you some basic instructions on how to start using the service.

Dependencies: The second node in the project structure, contains a reference to all the NuGet packages and class libraries the project relies on. These dependencies are split into four categories:

- Analyzers: These dependencies are related to code analyzers. These are the tools that you have installed to help you to make your code better. These analyzers can be thought of as linters that trigger at

compile time to help you avoid code smells. Each analyzer you install will validate your code by checking it against a set of rules. If your code does not adhere to a rule, the compiler will throw a Warning or an Error during build time.

- Frameworks: This folder contains a list of frameworks your project depends on. The information contained in this node is useful if you are publishing your website as a runtime-dependent. In those situations, all the frameworks listed in this node must be installed on the host machine in order for it to function
- Packages: Most .NET developers will be familiar with this node. It contains a list of all the NuGet packages that the project uses. If an installed package contains additional dependencies, these will also be listed as sub-nodes. You can remove any package from this node, using right-click & Remove.
- Projects: The last node contains a list of other projects in the solution the project depends on. You can add a reference to another project here, by right-clicking on the node and selecting "Add reference".

Properties: This node contains a list of references to configuration files used by the project. When the project is first created, a settings file called `launchSettings.json` will also be generated. Whenever you run your website in the Visual Studio debugger, the settings from `launchSettings.json` will be used to configure how the website is launched. Settings like the `applicationUrl` and `sslPort` port are found here.

Additionally, if you create a Publish profile, the corresponding config file will be created here within the `PublishProfiles` folder.

wwwroot: The`wwwroot` folder should be used to add any static files that your website uses. Add any assets, CSS files, or JavaScript files here. Umbraco also uses this folder to upload any images that are added within the media library within the CMS. These CMS assets can be found in the subfolder called `media`. All files in this folder can be directly accessed from the website. Say you had this image:

```
wwwroot > media > image1.jpg
```

You could access it using this URL:

```
http:///mywebsite/media/image1.jpg
```

App_Plugins: Within a Umbraco project, the `App_Plugins` folder is mainly used to store the files related to any third-party community Umbraco packages that have been installed. For example, if you installed the popular uSync package via Nuget, the files required for uSync to function will be added here.

Umbraco: This folder is used by the CMS. If you need to find a Umbraco specific file, start by looking in this folder. Within this folder, you will find lots of useful files. You will find the application logs, resource files used by

the CMS, the initial database files, all temp data including the files for the front-end cache and the Examine indexes and a lot more. This is also the default creation area for the Umbraco Models builder.

Views: The `Views` folder is the area that contains the Razor views. Whenever a new document type is created within the CMS, the corresponding HTML layouts are found here.

Asides from the web and system folders, an ASP.NET Core based project will also contain three key files:

AppSettings.json: In .NET Core, the beloved `web.config` has been retired. Instead, you now use `appSettings.json` to configure your application. In order to apply environment-specific settings, environment variables are used to tell the application to load additional configurations. The precedence in application settings loading order is:

- Environment variables
- Configuration settings

Application initialization and configuration

Up until this point in the chapter, it would have made no difference which version of Umbraco you picked. You could be using Umbraco v9 or v10 and the installation process would have been the same. It also would not have mattered which version of .NET you used. The steps you followed would have been the same regardless of ASP.NET 5 or ASP.NET 6.

The only major area where there is a noticeable change between the versions is within the files that are responsible for bootstrapping and configuring your application. These files are also fundamentally different compared to a Framework project. To help me get your head around these changes, let us start with a brief history lesson.

Within the .NET Framework, we had the ASP.NET Application File, better known as the `global.ascx` file. The `global.ascx` file used to be the entry point for the website. After a request came into the web server, the server would call the .NET handler. The handler would start by triggering an application starting event that the `global.ascx` would deal with. Within a Umbraco website, the `global.ascx` would inherit from a special base class called `UmbracoApplication`. The code that was responsible for booting up Umbraco resided within `UmbracoApplication`.

This process has completely changed within a .NET Core application. Up until ASP.NET 5, the files that were responsible for bootstrapping the website were called `Program.cs` and `Startup.cs`. `Program.cs` was responsible for starting the application, while `Startup.cs` was responsible for configuring the application.

To make life a little more confusing within ASP.NET 6, there have been two additional changes to these files. This is the only major area where depending on which version of Umbraco you are using will influence how

you structure something. Asides from this area, every other file will be the same.

The first big change is that `Startup.cs` has been deprecated within ASP.NET 6. Within ASP.NET 5 you had to add your application configuration code within `Startup.cs`. This is not the case within ASP.NET 6, within 6 you can now add the same code directly within `Program.cs` instead. Microsofts thinking behind this move was to make the application code simpler, however, when switching from Umbraco 8 to 10 it definitely complicates things!

I am going to cover `Startup.cs` in detail shortly, however, if you are using Umbraco 10 you should be aware that you do not need to use this file anymore. When you install Umbraco 9 `Startup.cs` will definitely get created and when you install a later version it may or may not.

Personally, I still think it makes sense to use a `Startup.cs` within Umbraco 10. Without using `Startup.cs` your `Program.cs` will become cluttered and you will find it harder to find the code you need. The main takeaway is that within Umbraco 10, everything that I mention that can be done within `Startup.cs` can also be created directly within `Program.cs`. Throughout this book, I will assume you are using `Startup.cs`, however, the choice is yours.

The second change is related to how code can be structured within `Program.cs`. ASP.NET 6 supports something called minimal APIs creation. In terms of how you write code within ASP.NET 6, the minimal API specification is a pretty big change. The specification allows developers to create classes in a more lazy way. A class created using a minimal style looks more similar to a file created in a Node or Javascript project, rather than a C# project.

When creating a minimal class, you do not need to include a class declaration or a namespace definition. An example of a minimal `Program.cs` file is shown below:

```
using Microsoft.AspNetCore.Builder;

var builder = WebApplication.CreateBuilder
(args);
var app = builder.Build();

app.Run();
```

Program.cs in .NET 6

As you can see from the example shown above, this new style allows developers to just get on with writing code. It does not require a namespace or class deceleration to be included, which to me looks very odd to see in .NET. If you compare how `Program.cs` is structured in ASP.NET 6 compared to within a Umbraco 9 website, you will see a noticeable difference:

```
public class Program
{
    public static void Main(string[] args)
        => CreateHostBuilder(args)
            .Build()
            .Run();

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(x => x.ClearProviders())
            .ConfigureWebHostDefaults(
                webBuilder => webBuilder.UseStartup<Startup>());
}
```

Program.cs in .NET 5

From a quick glance, it is instantly noticeable how different the structure between the files has become. For the rest of this book, I will assume that you are using the original way of writing C# classes everywhere except for `Program.cs`. The original way of writing classes in .NET is never going to go away and the majority of developers will carry on writing code the way they always have!

Program.cs: In Umbraco 10 and 9, `Program.cs` is now the entry point for the application. As you have seen, under the hood what used to be a web project is now constructed as an application. Like all .NET console applications, the application needs to be started before it will start to serve any of your webpages. Within ASP.NET 6, the server will simply called `Program.cs`. Within `Program.cs` you will need to initialize a new `HostBuilder` object. This object will allow you to start the application. After initializing the builder object, you are free to add whatever configurations you like to it. When you are happy the application has been configured the way you would like it, you start the website using the `Build()` command.

If you are using Umbraco 9 this process is slightly different. In V9, the server will call `Program.cs`, however, in .NET 5 world the framework excepts to find a method called `Main()`. Within main you would then add the code to initialize the `HostBuilder` object. The overall process is very similar just the implementation code is slightly different.

Assuming, you are using a `Startup.cs` file, you will need to run all the code within `Startup.cs` before you call `Build()`. THe code to do that is shown below:

```
var builder = Host.CreateDefaultBuilder()
                  .ConfigureUmbracoDefaults()
                  .ConfigureWebHostDefaults(webBuilder =>
                  {
                    webBuilder.UseStartup<Startup>();
                  })

var host = builder.Build();
host.Run();
```

Program.cs calling Startup.cs

As a minimum, this is all the code that you need to bootstrap your website. As the book progresses, you will learn about some of the more advanced capabilities that you should enable within this file. Within a real-life `Program.cs`, as a minimum, I would expect to see some code that configures the logging framework as well as some code to tell the framework how to access environment/developer-related configuration.

Within the .NET Framework, we could read environment-specific configuration using different files, like `web.release.config` or `web.staging.config`. We can do exactly the same within ASP.NET 6 using `applicationsettings.json`. The difference between Framework and ASP.NET 6, is that you will need to write some code to make it work. Within ASP.NET 6 that code will be located here. More on that later on though!

Startup.cs: Within ASP.NET 6, `Startup.cs` is the file where you can add the code that configures what happens when your application starts. If you are using ASP.NET 5, then `Startup.cs` is the file where you have to add this code. `Startup.cs` can loosely be compared to the `Global.asax` file within ASP.NET Framework. It is within `StartUp.cs` where you will be adding all the code that configures how your application should run. In v9, this is the file where you will find all the code that is required to load Umbraco correctly. In applications using Umbraco v10 and above, you may find the Umbraco boot-up code here, otherwise, it will be in `Program.cs`. The `StartUp.cs` class has three main purposes:

- Run all application initialization tasks
- Register the dependency injection framework and the services
- Define and register all the middleware required to run the site

All .NET Core applications require a `StartUp.cs file`. `StartUp.cs` has two methods `ConfigureServices` and `Configure`.

- `ConfigureServices` is the method to configure the IoC container. You can still configure the container in a Umbraco composer if you want to, however, in terms of .NET core this is the new recommended place to configure dependencies.
- `Configure` is the method to configure the request pipeline. It is within `Configure` you register any middleware used by the application

An example of a vanilla Umbraco `Startup.cs` is found below:

```
public class Startup
{
    private readonly IWebHostEnvironment _env;
    private readonly IConfiguration _config;

    public Startup(IWebHostEnvironment webHostEnvironment, IConfiguration config)
    {
        _env = webHostEnvironment;
        _config = config;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddUmbraco(_env, _config)
            .AddBackOffice()
            .AddWebsite()
            .AddComposers()
            .Build();

    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseUmbraco()
            .WithMiddleware(u =>
            {
                u.UseBackOffice();
                u.UseWebsite();
            })
            .WithEndpoints(u =>
            {
                u.UseInstallerEndpoints();
                u.UseBackOfficeEndpoints();
                u.UseWebsiteEndpoints();
            });
    }
}
```

Startup.cs

Notice that within `Configure` a call is made to `UseUmbraco()`. This is the code that bootstraps Umbraco. All the routing rules, composers, and back-office shenanigans that the CMS requires in order to run are triggered from this line of code. For comparison, within v8 this registra-

tion code was called from within `UmbracoApplication` which was part of the `global.asax`

Now you have a good understanding of the default project structure, let us think about how to optimally add your custom code within this structure.

Class Libraries

Before you start creating controllers, models and views, you should take some time to consider where you want to add that code? How you structure your project will be based on your preferences. Design and architecture are subjective topics. If you have strong opinions on how you want to structure your project, great, skip this section and do what makes you happy. If you are new to Umbraco or you simply want to adopt my approach, I will outline it below.

My recommendation is to avoid adding custom code to the main website project. As a minimum, I recommend that you add all your custom code into at least one custom class library. Over the years I have changed my appraoch to project structure a few times.

I used to structure my custom code into class libraries in a very granular way, adopting a more design-driven domain architecture. From my experience, in terms of refactoring, releases, ease-of-working etc… the biggest benefit was having code separated into at least one class library. Anything after that, you get marginal gains.

The benefit of having all custom code in a class library is the ability to rip out the website project easily. I have always found that if you need to upgrade from v6 to v7, or, v8 to v9, having your code separated from the website project makes these upgrades easier. When it comes to writing unit tests, having your code in a separate library makes writing tests a little bit easier, however, the bit benefit for me has always been around upgrades.

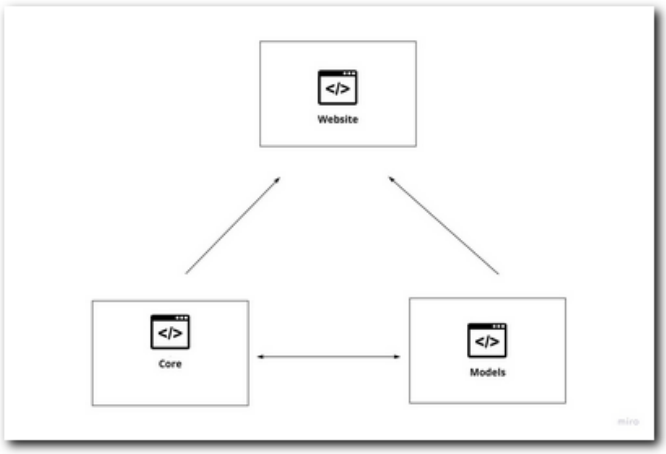
Nowadays, I tend to favour having one class library to store all my project related code and one library to store any code I might share between projects. A typical project structure on one of my projects would look like this:

- `Project.Website`
- `Project.Core`
- `Umbraco.Models`

You may notice an additional library here called `Umbraco.Models`. This library will be used by Umbraco as part of its CMS auto-generation process. The model builder will convert document types created by content editors within the CMS to C# models that can be used in code. We will come back to this topic in more detail later, for this discussion these models will impact how these additional class libraries are referenced.

A slight nuance with additional class libraries is accessing the models generated by the Umbraco models builder. If you want to access a model generated by the model's builder within one of your class libraries, you will need to change the default model generation behavior.

The reason for this is to avoid a cyclic reference error. If your class library references the website project to get the models and the website project references the class library to access the custom code, the compiler will get into an infinite loop. This is why you will need to create a separate `Umbraco.Models` class library if you follow this architecture. Following this pattern your project structure and the references look will look like this:



Umbraco Package Browser

In order to setup your additional class libraries so they have the correct dependencies to work with Umbraco, you will need to ensure that they reference the correct NuGet packages. Below gives a brief summary of each library, summing up its purpose and detailing what packages it will need to reference:

Project.Core: Add all custom code here. All controllers, view models, composers, components, view components, dashboard, etc.. get created here. As you be adding Umbraco related code within this project you will need to ensure this library references these NuGet packages:

- Umbraco.Cms.Core
- Umbraco.Cms.Web.Website
- Umbraco.Cms.Web.Common
- Microsoft.AspNet.Mvc
- Microsoft.AspNetCore.Mvc.Abstractions

Umbraco.Models: This library is solely used to store all the models generated by the Umbraco model builder. In order for the project to compile after you generate the models, you will need to reference these Nuget packages:

- Umbraco.Cms.Core
- Umbraco.Cms.Infrastructure

When it comes to content modelling within the CMS and working with that data in code, your life will be much easier if you can work with C# representations of those items. This is what the out-of-the-box Model Builder will provide.

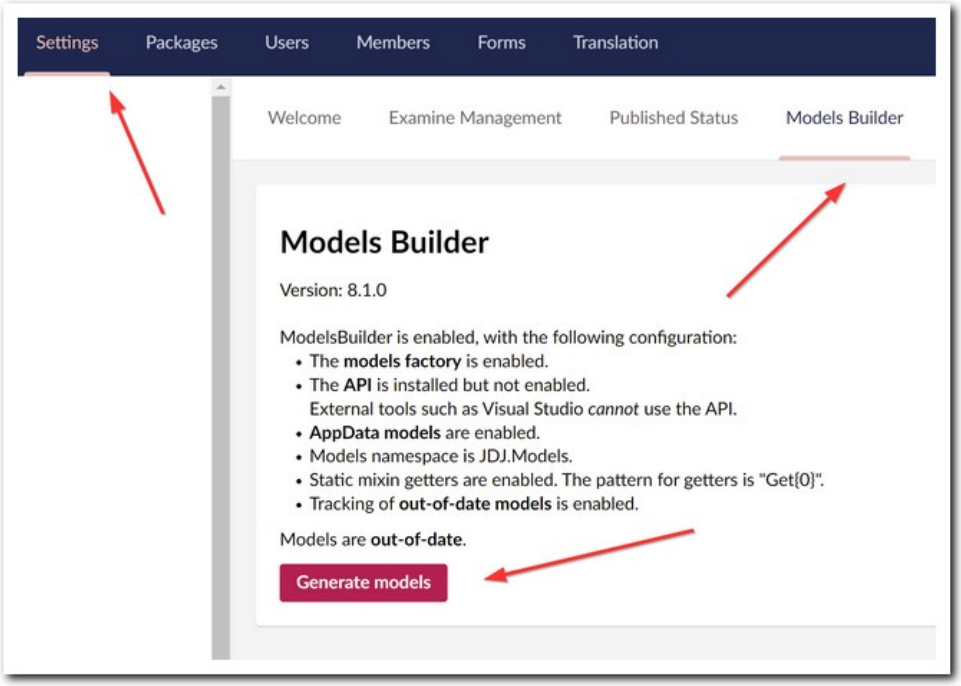
In terms of code generation based on CMS data, Umbraco uses a model first approach. I will cover this in more detail later, for now, the important takeaway is that Umbraco can generate C# classes based on the document types created within the CMS. You can then use these models in your code to more easily interact with the CMS.

The only caveat for using these models within a custom class library is this cyclic issue. To get around it, you will need to change the default behavior of the Builder to generate the models outside of the 'Website' project. The ModelBuilder is configured within `AppSettings.json`. To ensure your application will generate the files outside of the web project and instead create them within a class library called `Umbraco.Models` you will need to change the configuration to this:

```
"ModelsBuilder": {
    "ModelsMode": "SourceCodeManual",
    "ModelsDirectory": "~/../Umbraco.Models",
    "AcceptUnsafeModelsDirectory": true
},
```

There are three important settings to understand here:

ModelsMode: Set this to `SourceCodeManual`. This setting means that Umbraco will only generate models when someone clicks the `Generate` button within the CMS. You can find this button within the `Settings` section:



Generate Models With Models Builder

ModelsDirectory: The folder location where the models will be generated. The setting will be read from the Website project, so you need to move back one directory `~/..`, then list the name of the folder where you want Umbraco to generate the models.

AcceptUnsafeModelsDirectory: This setting needs to be enabled in order to allow the models to be generated outside of the Website project. If you need to use `../` within the `ModelsDirectory` key this setting needs to be set to `true`

This configuration will allow you to use ModelBuilder within the additional class library. This is not the only way to configure this package. You can learn about all the other ways of configuring Model Builder here.

.gitignore

I assume that you are using source-control. Just like on a normal vanilla ASP.NET build, it is not recommended to commit every folder within your project into source control. There are also several Umbraco specific folders that you will want to exclude as well. These files and folders will add no value from being committed, and will only add bloat and delay build times.

When using Git you should always create a '.gitignore' file in the solutions root directory. Below lists all the folders I recommend that you add to the gitignore' file:

- ***/bin*** - Assuming you use Nuget to manage all the project references, then this folder should be excluded.
- ***/wwwroot/media*** The 'Media' folder is where all the images and files that are uploaded via the Umbraco backend are hosted. If you add these into source-control things can get large quickly! Also, if use web-publish to deploy code you need to be mindful that you may override the live files by accident. Backing up the media folder is very important, however, source-control might not be the correct solution.
- ***/obj*** This is the .NETs temporary storage folder. It is the folder that is used when the build pipeline is compiling your site. No custom files should even be stored in here
- ***/umbraco*** This is where all the core Umbraco files are located. Unless you have a good reason, none of these files need to be added into source-control

Useful Developer Tweaks

Think about it, during a new CMS project how many hours a day do you spend crafting code? Instead of wasting your time on admin tasks, it makes a lot of sense to optimize your solution and your workflow so you

can just concentrate on writing code. In order to boost your productivity, you need to relentless eliminate all unneeded steps within your edit, compile, debug cycle. In this section, I will cover some non-essential but handy tweaks that you to spend more time focused on code.

Configuring Local Developer Connection Strings

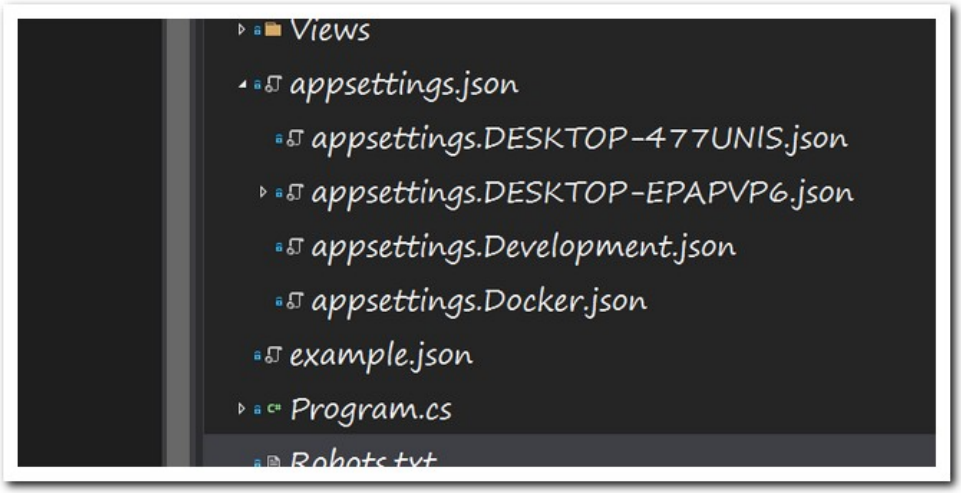
More often than not, multiple developers will be working on a project. Each developer will want to set-up their local environment to suit their individual workflow. This means that on a project, each developer will likely want to use slightly different project settings. These settings could include folder locations and connection string details.

For any developer working on a project, a potentially big time suck is having to continually apply these changes on an ongoing basis manually. Some teams might decide to add config files into their `.gitignore`. Each team member is then responsible for creating their own config file and managing it locally. This works but is easy to lose your settings. If you need to perform a `git clean -f` or a `git reset --hard` your config file can be lost! To speed things up, some developers might create a stash that contains their local changes and then pop and push those changes every time they do something with origin. Having to manually apply these changes whenever they need to pull or push changes into source control is a pain. Some teams might simply choose to force everyone on the team to work with the same settings, using a shared database and agreeing to use the same folder location on each team member's local PC.

One way to reduce this type of project admin friction is to automate this process. On a project, I want to make sure that it is super simple, quick and automatic for individuals to apply their own settings. The added bonus of dealing with this issue is that it will also make it really easy for someone new to the team to get up and running with it.

To solve this issue, the solution that I recommend you adopt is based on creating a separate config file per developer. In a ASP.NET Core based website, each developer will create a custom `appsettings.json` file that is unique to them. The best way I have found to do this is to use their machine name within the filename. If your PC is called `spotty`, the individual would create a settings file called `appsettings.spotty.json`. Using a build task, you can make sure their individual changes get applied whenever they try to build the project.

The good thing about this approach is that all your config files can be checked into source control. Following this approach, you will add config files like the example below:



Umbraco Starter Kit

To enable the application to read these configuration files on startup, you will need to add some code within `program.cs`. Within `CreateHostBuilder` you can add code that will tell the framework to apply any changes found in a local fine if one exists. The code to do this looks like this:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(x => x.ClearProviders())
            .ConfigureAppConfiguration((ctx, builder) =>
            {
                builder.AddJsonFile("appsettings.json", false, true);

                builder.AddJsonFile($"appsettings.{Environment.MachineName}.json", true, true);

                builder.AddEnvironmentVariables();
            })
            .ConfigureWebHostDefaults(webBuilder => webBuilder.UseStartup<Startup>());
}
```

CreateHostBuilder()

This code will first load the settings found in `appsettings.json`. Notice, the additional parameters that are passed in to `AddJsonFile()`. The second argument tells the compile if the file is mandatory or not. `false` means that it is required, `true`, means optional. In the second call to `AddJsonFile()`, the code looks for a config file containing the machine name. If an `appsettings.MACHINE_NAME.json' config file is found, override the values found in the default settings file with any settings found in this new file. As this second line is optional, if no file exists, no big deal!

Following this approach will mean that when you need to onboard a new developer, all they need to do is install the website's database locally, create a config file with a valid connection string and then commit the file. Every time they build the site, their own settings will be applied. This is a

very simple and robust way to fix the local developer configuration problem.

Automatically Include All Files In A Folder

One annoying Visual Studio quirk is that by default, the tool will not automatically include new files added within a folder to the project. Within a Umbraco project, this means that newly generated models created by the Umbraco ModelBuilder will not be included within the project. Whenever you generate a new model, you need to remember to add that model manually to the project. The classes will be generated within the `Umbraco.Models` folder, however, you will not be able to write any code that references it, until you include it.

To make your life easier and rectify this pain point, Visual Studio can be configured to automatically include such files. Applying this change is done within the libraries `.csproj` file. This tweak needs to be applied to an empty library. If you have already generated some models and referenced them in your library you need to remove them first. To do this, select all the models, right-click on them in Visual Studio and select `Exclude From Project`. With the models gone, open the `ModelBuilder.csproj` file for the library within a text editor. Next, add the below config under the `</Project>` section:

```
<ItemGroup>
    <Content Include="**" />
</ItemGroup>
```

Next, close the solution within Visual Studio and re-open it. Browse to the `Umbraco.Models` library within Solution Explorer. All of the models should now be automatically included. Whenever you generate a new model they will also now magically appear here. Another manual step eliminated!

Another area where this config change can be handy is the Media folder (`wwwRoot > media`). Applying this tweak will depends on your policy around checking web assets into source-control. If you need to publish your website frequently, adding assets to the project will slow the build and publish process down. If you are working locally and you only have a handful of assets, this tweak can help speed up your development especially when you are using containers! To allow for Umbraco media library items to be automatically added to your project, follow the process mentioned above. Remove existing files first. Next, add the below config within the website projects `.csproj` file:

```
<ItemGroup>
    <Compile Include="wwwroot/media/**" />
</ItemGroup>
```

All new media items will now be included within your project.

Further Reading

- How To Install Umbraco 9 And Setup A Development Environment
- Install Umbraco V9 In Visual Studio (And Debug Some Issues
- Essential Developer Tweaks To Make In Umbraco V9
- Overcome Publish Locking Errors In Umbraco 9
- Locked Files When Publishing NET.Core Apps
- Publishing And Running ASPNET Core Applications With IIS
- Browser Umbraco StarterKits
- Should I use self-contained or framework-dependent publishing in Docker images?
- Learn How To Host An Umbraco 9 Site Using Docker
- Umbraco 9 Models Builder Deep Dive
- How to persist data in Docker: Volumes
- Cyclic reference of projects not allowed
- Umbraco release cadence

Content Modelling

When undertaking a new Umbraco project, the first major task that the development team will need to tackle is content modelling. Content modelling is the process of taking the websites wire-frames and/or the designs and breaking them down into CMS components. The end result of the content modelling process is a complete set of CMS templates and components that content editors can use to build a fully functional website.

You will need to decide if you want to model the whole site in one go, or if you model-as-you-go in parallel with development. There is no hard or fast rule on which route is better. Personally, I typically model the complete site at the start of the project. I find that having a more complete and in-depth understanding of all the tasks required to complete the project as soon as possible is very beneficial for planning.

When you have a more complete picture of your project, it is that much easier to plan out sprints and create backlog items. After you have a rough plan of the steps required to complete the project, broken down into sprints, it is easier to understand the timelines and plan the launch date. All projects will have a sponsor that you will need to keep happy. Having a clear plan that the wider team can view will help aid in that communication.

Every project will have a deadline. A lot of the deadlines that you will be asked to work towards during your career will be tied to some specific event or business critical milestone. These deadlines must be met no matter what. In a lot of cases, a deadline might simply be plucked out of the air from someone in the business. I recommend that you learn which category your project deadline falls into.

You can only have high confidence about a Umbraco project's estimate after the whole site has been modelled. Without opting to perform the full content modelling process at beginning of a project, you can never fully trust how realistic hitting a deadline will be.

After going through the content modelling and planning process, you may realise that it is impossible to reach a deadline. If you discover the deadline is impossible to hit, you can warn the project sponsors much sooner. From my experience, having conversations about extending the deadline or cutting scope tends to go more smoothly the sooner you can have them. Do not have this conversation a few days before the planned go-live, otherwise, things can become…. tense!

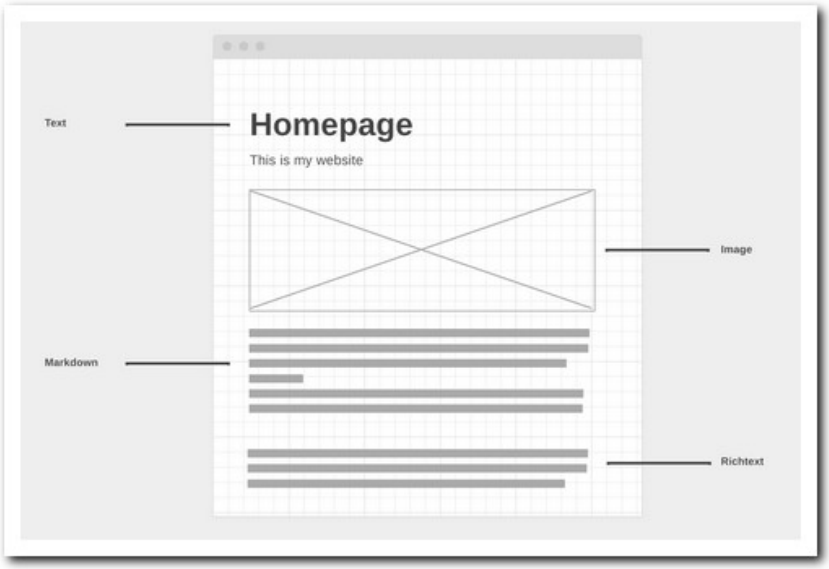
When it comes to content modelling, every project will be unique. Sadly, there is no one size fits all architecture that will work optimally on all sites. There is no single pattern that will solve all content challenges. It will be up to you to decide how to configure the backend so it makes the most sense. This is after all why you get paid the big bucks!

In a Umbraco project, it will be your responsibility to take design and then decide what Umbraco document types will need to be created. For each document type, you will also need to identify the properties it will need to expose. After you understand what needs to get created, it will be up to you to then build the site.

Content modelling is the process of converting a design into a bunch of templates. For each template, you will need to further deconstruct the content on the page into basic elements like “text”, “image”, “date” or “location”. During a content modelling phase, you will need to optimally break down the content into sensible chunks for authoring. For each chunk listed within the wire-frames, you will need to decide what will be the best mechanism to allow a content editor to add content into the document type so that it adheres to the design.

There is an art to successful content modelling. There will always be a trade-off between reusability, localization, enforcing UX, and even creativity. Should you give the content editors ultimate flexibility to make content decisions but run the risk the site looks disjointed? Should you instead build a system where everything is locked down to preserve the integrity of the designer’s vision? There is no right or wrong answer to these questions. It will be up to you to decide what you think will work for your project. This is the hard part of content modelling. Experience in this area can definitely help to make your project work. If you are lacking experience, fake it till you make it!

Just like how Neo is able to see the code in the Matrix, after you have been content modelling for a while, whenever you see a wire-frame you will automatically see it in terms of underlining components. Let us prove this theory with an example:



Wireframe Example

The wireframe above depicts a basic Homepage. To model this wireframe within Umbraco, you would create a document type with an alias called `Homepage`. Even though there are only a few elements on the wireframe,

there are a surprising number of ways you could model those elements within Umbraco. One possible permutation could be like this:

- A property called `Title` of type string
- A property called `Banner Image` of type Media picker
- A property called `Main Content` of type Markdown
- A property called `Footer Area` of type Rich Text

Another way would be

- A component called Call To Action
- A property called `Main Content` of type Rich Text

The main takeaway from this very simple example is that there are multiple ways to model a single wireframe. Going through the content modelling process will make you focus on the finer details and practicalities of how you will build your site and how the content works together.

When modelling a page, I recommend capturing all the details within a structured document. As mentioned in my Umbraco v8 book, I add all these details within a document I call a component catalogue. The component catalogue can be thought of as the developer's project bible. For each page and component that I uncover within the wireframe, I create a corresponding component within the catalogue. For each element, I add these details:

- A descriptive name that will be displayed within the CMS
- A description of its purpose
- The properties contained on the page or component

For each property, I further define:

- Whether the property is compulsory or optional
- Whether the data comes from an API or the CM
- Any validations that need to be applied, like the word count or character limit

Rinse and repeat this process for every wireframe and design. Afterwards, you will have a complete picture of your website.

I do not recommend teams to try and start the content modelling process prematurely. The optimal time to start modelling for a development team is when launch day sign-off has been agreed upon. Launch day sign off is based on two aspects. The marketing team agreeing at a high level on what content will be required. The second is the business signing off the designs. Failing to get this sign-off before you start modelling only increases the risk of inefficiency.

Once a content model has been defined and created within the CMS, every subsequent change will require a developer to update it and fix any broken code as a result. The more frequent a model changes, the less productive the development team will be.

Content Modelling Within Umbraco

Successful content modelling is a key aspect of every Umbraco project. Remember, you are defining the content authoring experience. This experience will be used for months, or maybe even years. How good of a job you perform in this phase will determine how convenient the backend will be for content editors to use. This impact could potentially last for the entirety of the site's life span. Get it right and the editors will find it very easy to create and edit pages. Get it wrong and suffer their wrath!

Content modelling is usually performed based on the website's wire-frames. In the wire-frames, you will identify types of pages. In Umbraco terminology, these types of pages are called document types. Document types are composed of properties. The gotcha when modelling properties within Umbraco is that a property is not a single thing. Within Umbraco, each individual property is constructed of three components:

- A property definition
- A data-type
- A property list editor

The good news is that on a day-to-day basis, you do not need to understand these nuances. The process of creating properties using the UI is intuitive and straightforward. As we want to become Umbraco masters, it is handy to understand the differences. This is why we will cover these topics in more detail later on.

Document types are the bridge between content and code. This bridge means that document types are the pillars of the content modelling process. Each document type will define a schema that a content editor can use to create pages from. Document types define how properties and content are arranged and displayed to your editors.

The document type definition does not store or contain any content itself. Conceptually the process is the same as how a class is used to instantiate an object. Once you define the schema, an editor can create an unlimited amount of pages from it.

Document-types are defined within the Umbraco backend from the `Settings` section. The `Document Types` node can be found within this screen at the top of the left-hand menu. From this area, you can create a new document-type by clicking on the three ellipses next to the `Document Type` node. From the context menu that should appear, select `Create`. From the `Create` screen you have five options:

- Create Document Type With Template: Creates a document-type, a template with a corresponding `cshtml` view and maps the two things together within the CMS. As you will shortly see, a template within Umbraco refers to a UI file that contains the pages HTML. This is a good example of a common industry-specific term like

`template` meaning something different within a framework vernacular.

- Create Document Type: Creates a document type without any of the UI stuff.
- Create Element Type: Creates a document-type that can be used as a component rather than a page. More on this later!
- Create Composition: Creates a re-useable component. Compositions can be used to model things like shared settings.

When creating a new document type, there's one mandatory property that you need to define before you will be allowed to create the type. There are also some optional best-practice fields you should also fill in, these are:

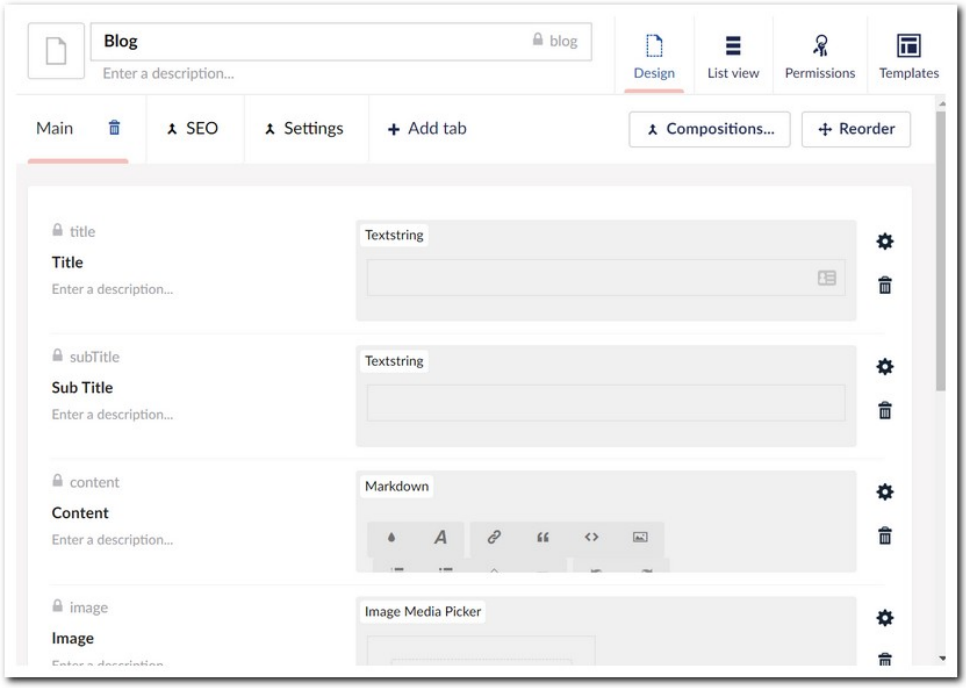
Name: The name defines the user-friendly reference that content editors can use to identify that type within the CMS. When the document-type is saved, a corresponding `alias` will also be created. The value within the `alias` field is the value that you will use to reference the document-type in code.

When you save the document type, an `alias` will be generated. On save, Umbraco will use the value added within the `Name` property to generate this alias.

The alias is the property that you will use to reference the document-type in code. During this conversion process, Umbraco will convert the `Name` property using camel case. The conversion process will also strip out any illegal characters. SO just be aware the `Name` and `alias` will not always 100% match each other!

Description: The description will help content editors to understand the purpose of the document type and when it should be used. This is an optional field, however, I recommend that you always include a description to improve the experience for content editors.

After creating a document-type you will be redirected to the editing page. The editing page is where you will perform the content modelling for that type.



Create A New DocumentType

The editing screen is composed of four tabs:

Design: This is the screen to add, edit, re-order, or remove properties. From this screen, you can also create tabs to make the document type easier to use for the content editors.

Listview: This tab contains a single property that when enabled can help content editors find child pages more easily. When enabled, this feature will create a new tab that will render a list view component. This list will be populated with any pages that have been created underneath the current page. Having this list tab can help content editors find content within the backend more easily. For example, imagine there are over one hundred child pages underneath a blog listing page. Content editors will be able to find content that much easier using a list that can be filtered and searched, compared to trying to find it manually within the content tree.

It is considered good practice to enable this feature on any document type that will be used as a hub or listing page. If you know the editors will be creating lots of child pages underneath this type of page, enable it.

Permissions: Personally, I think this tab would make more contextual sense if it was called `Settings`. This tab will allow you to configure several key things on how the type will behave:

- Allow As Root: Enable this feature if you want a content editor to be able to create a page of this type directly under the root node. You will definitely want to enable this setting for the homepage. You will likely want to disable it for all other types.
- Allowed child node types: This feature defines what types of pages a content editor can create underneath a page created using this type. When this feature is used the creation list will be filtered to docu-

ment types specified within this list. Use this feature to make sure editors do not create pages in areas that it was never intended to be used!

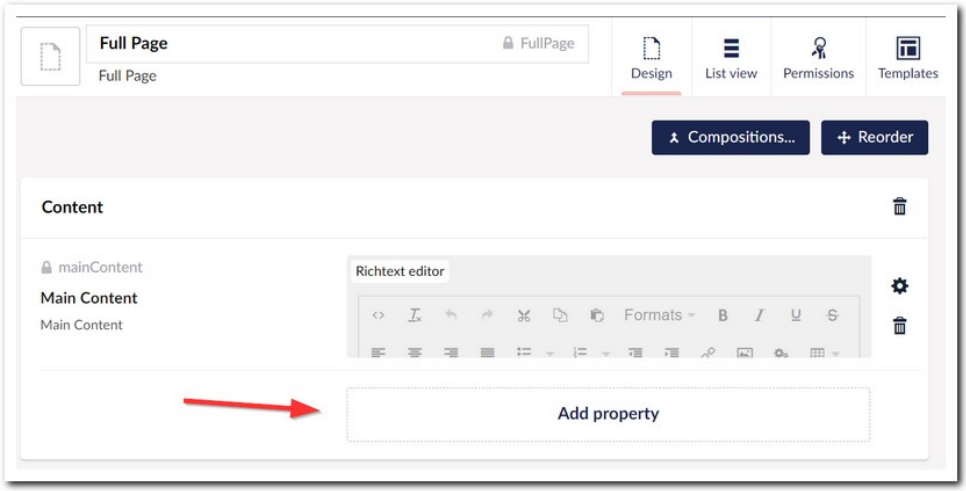
- Allow vary by culture: Allow the document-type to be used with multi-languages
- As an Element Type: Document-type set as elements can be used as components rather than pages. Umbraco will not allow content editors to use these types of document types within the page tree. Element types are useful for dynamic content creation. We will cover this in a lot more detail later.

Properties, Data Types and Property List Editors

The next fundamental part of the content modelling process is assigning properties to document-types. In order to optimally model a website, you need to know which tools you have at your disposal and maybe more importantly how they work. You will probably find that the conceptual mechanism of how you assign properties to document types is less intuitive than you might imagine.

In most CMS systems, a property is created as a single thing. In essence, you add a the property onto a template, job done. Within Umbraco, what you would traditionally label a property, is built using three components, a property definition, a data type and a property editor.

A property definition is the mapping of the property onto the document type. You can add a property onto a document type from its editing screen. Select the `Add Property Button`.



Adding A New Property

Clicking on this button will launch the `Property Settings` dialogue. From the `Property Settings` dialogue, you can define a new property

and its meta-data. To successfully add a new property onto a document type, you will need to include this metadata:

Property Title: The friendly name that will be displayed within the CMS. This is a mandatory field

Property Editor: The property editor field is used to define the type of property that will be associated with the property. Despite the label, this field is used to attach a data type onto the property rather than attach a property editor directly. The data type defines a mapping between a property editor and the property. When you are starting to learn Umbraco, understanding the difference between a data type and a property editor can be confusing. This is why we will deep dive into the differences in more detail shortly. The property editor field is also mandatory.

Validation: This field determines if the property will be mandatory or optional. The default option is optional.

Custom Validation: It is also possible to apply additional validations onto a property. Out-of-the-box, you choose to validate for an email, a number and a URL. You can also add in your own custom regular expression.

Appearance: Defines the position of the label that will be rendered in the CMS relative to the property

The most important field within this list is the property editor. The property editor field determines the type of data that a content editor can create using that property. A property editor defines all the fields that will be rendered within the page creation screen for that property. For example, the Label editor defines two properties. One for the content editor to define the label and another one called `Value Type`. `Value Type` defines the type of data that is allowed to be entered for the label. Default options include `String`, `Int`, `Time`, and `Decimal`.

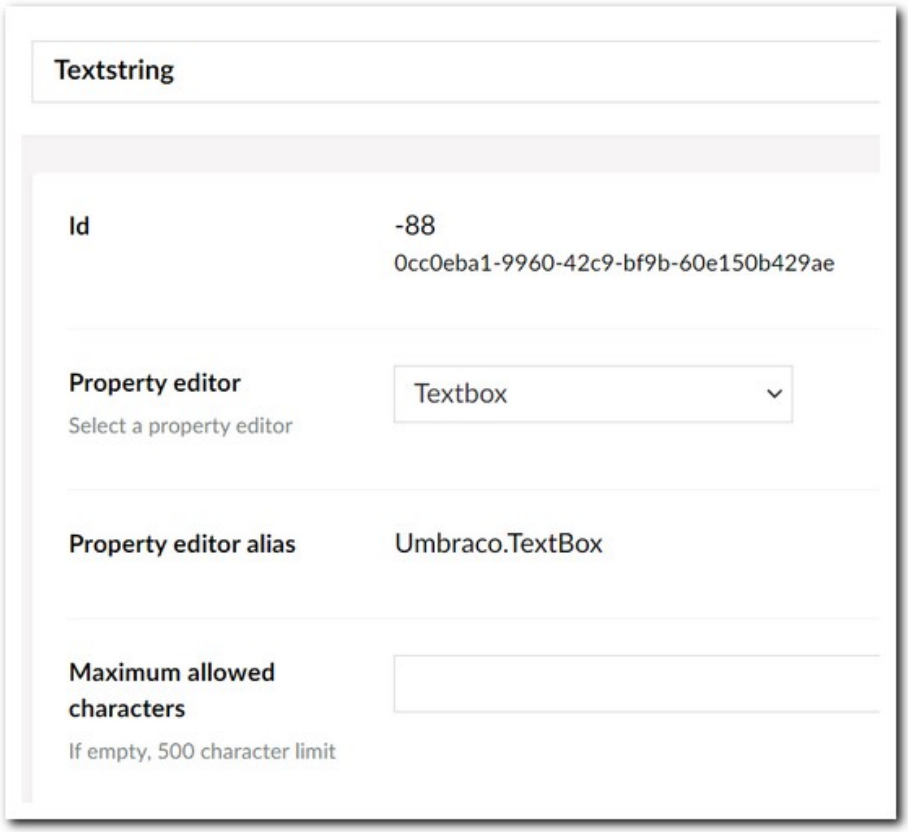
When picking the property editor you would like to associate with the property, you will be presented with the full list of property editors that are installed within your CMS instance. Examples of some of the default property editors include string, boolean, text-area, and a content picker.

To select the property editor that you would like to use, you click on the related icon from the property editor screen. This action will load a new dialogue. From this screen, you have the option to either create a new data type based on that property editor or, pick an existing data type. It is the data type that is associated with the property, not the property editor itself.

A data type is the mapping between the property editor and the document type. A data type defines the property editor the property will use, as well any specific configurations that should be applied. When creating a new data type, you can prefill in any of the fields defined by the property editor. In the label example mentioned above, you might create a data type called `Number Label` where the `Value Type` field is preset to `Int`.

Compared to other topics, conceptually data types are not as intuitive to understand. To highlight the role of the data type, let us walk through another example.

Out-of-the-box, Umbraco ships with a property editor called `Textbox`. `Textbox` defines a few configurable fields including a value field and a max limit field.



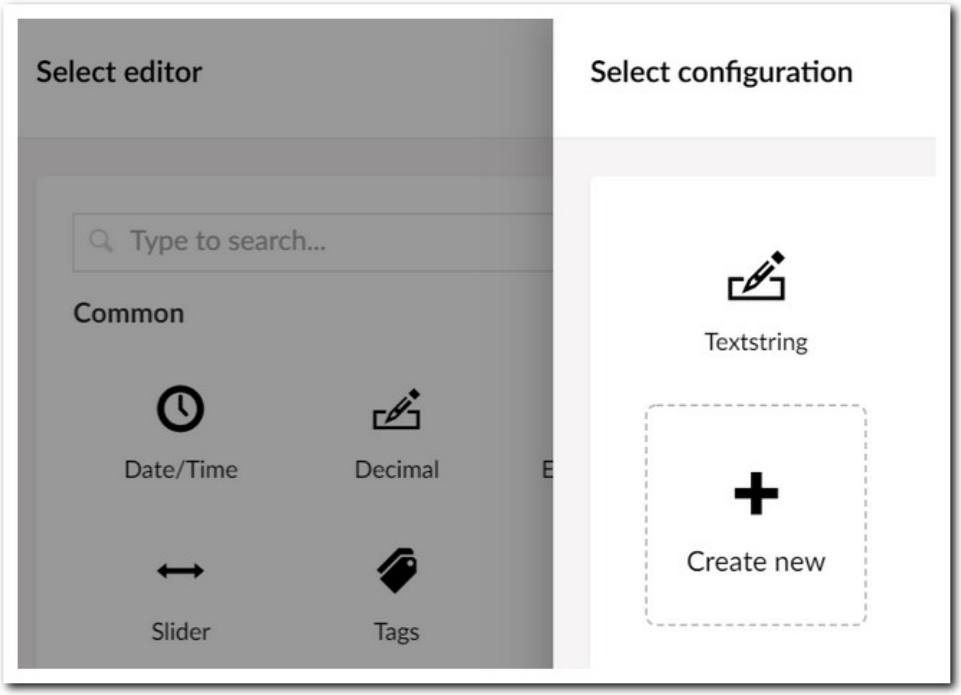
Textstring Datatype

Out-of-the-box, Umbraco also ships with a data type called `Textstring`. The data type named `Textstring`, by default is set to use the `Textbox` as its property editor. `Textstring` does not set any other default values. This means the max limit value field is left empty.

If you select the `Textbox` editor from the property editor selection list, you will have the option to select the existing data type called `Textstring`, or, create a new data type. If you decide to create a new data type, you can define the fields that the property editor exposes here.

Imagine that during the content modelling process, you notice a need to allow content editors to add a country code from a lot of different document types. A country code is comprised of a code of exactly 3 characters. To ensure good content hygiene, you want to enforce this rule. One solution to model this within the CMS would be to create a new data type called `CountryCode`. `CountryCode` could use the `Textbox` property editor as its default type. Within `CountryCode` the max limit field can be populated. In this example, it would be populated with the number `3`.

This `CountryCode` data type can now be more easily reused in any other document type.



Use Textstring Or Create New Datatype?

Whenever you want to apply a country code property onto a document type, you can select the existing data type. If a content editor ever encountered that property within the CMS, they would never be allowed to add more than 3 characters into it.

Granted all this talk on property editors and data types might sound confusing. When it comes to creating a property within the UI things are pretty straightforward. In pure content editing terms, I doubt you would even notice a difference unless someone specifically called it out. For us developers, understanding what this decoupling provides is important. Understanding these nuances will help you content model like a pro!

All the data types that exist within your CMS instance, can be viewed from the `Data Types` node within the `Settings` section.

In terms of Umbraco's best practices, it is recommended that you try to re-use data types wherever possible Trying to re-invent the wheel will only make the CMS more cumbersome to use. Granted on smaller sites, creating a few extra data types will not have a massive impact, however, it is always good to get into positive habits!

Property Editors

A property editor defines how the property will behave within the CMS. An editor defines and references the UI files that content editors will use to insert content into Umbraco. It will also define the mechanism of how that data is stored. A property editor is comprised of files, both client-side

(with HTML) and server-side (with C#). The HTML defines how the properties UI will look within Umbraco. The code files will define the data binding.

When you install Umbraco over 30 property editors will be installed. A thing to be aware is that a default data type will not be created for each editor. Property editors are not directly accessible via the CMS, only via a data type. This means that if you try to locate a screen called `Property Editor` within the CMS you will be out-of-luck!

A property editor is used as the base-type for a data-type. For the curious among you, you can find all the default views used by these out-of-the-box property editors within your webroot, here:

```
wwwroot > umbraco > views > propertyeditors
```

Having files located within the `view` folder, immediately tells you that Umbraco makes web requests from the backend to these files in order to render the properties within the `Content` section. If you inspect any of the these files, you will see they are constructed out of HTML and AngularJs. These files define how the property will look within the CMS, they do not define the logic. Within v9 or v10, the files that contain the logic are found in C#. The code that powers these editors can be found within this namespace:

```
Umbraco.Core > PropertyEditors
```

This means that most of the files related to the default property editors can not be viewed directly within your solution. This code can only be found by looking within the `Umbraco.Core` assembly.

When learning a new CMS, I recommend that everyone spends some time getting familiar with the CMS codebase. As Umbraco is open-source, its easy to clone the CMS source-code from GitHub to inspect locally. Even easier you can look at it online at GitHub. You can see all the code related to v9.4 property editors from here.

Out-Of-The-Box Property Editors

In terms of the number of different types of properties that ship with Umbraco, I have yet to find a CMS that supplies more. The amount of property editors that Umbraco supports is pretty amazing. 99% of your content modelling needs can be met by the default properties without the need for any customization. In this section, I will cover all these default types with an explanation about each editor's purpose and where it can be used.

Block List: The Block List editor property will define an editable area on a page where content editors can add pre-built components. The Block List editor provides content editors with the ability to create pages with dynamic layouts.

It is possible to create re-usable components within Umbraco. These components are also referred to as blocks. Examples of common blocks I tend

to create on most projects include a carousel, call to action, banner, sliding card, spotlight area, etc…

In terms of content modelling, the block list editor is very powerful and very configurable. For this reason, we will delve into this topic in a lot more detail in the next chapter. Blocks are modelled the same way as you would define a page, using document types. To do this, when creating a document-type set it as an `Element`.

Checkbox list: The checklist editor property allows you to define a list that content editors can choose from. The list can be configured to allow single or multiple value selections. The code to render this list within a view looks like this:

```
@foreach (string result in Model.MyCheckboxListProperty) {
   @result
}
```

Checkbox List Editor

On the page, Umbraco will render the data in this format:

```
["optionOne" , "optionTwo", "optionThree" ]
```

Color Picker: The color picker property will render a set of predetermined hex-colours that content editors can choose from. This property is useful if you want to give editors some flexibility on colour, but also need to enforce brand colours. This property is useful to allow editors to pick the background colour of a button or section on a page. The code to render this picker within a view looks like this:

```
string result = Model.MyColorPickerProperty;
```

Color Picker

On the page, Umbraco will render the data in this format:

```
#32a852
```

Content Picker: The content picker property will allow a content editor to pick a specific page from the content tree within the CMS. This property is useful if you want to create a dynamic link on a page and you want to provide a content editor with the ability to update the link themselves.

The property will return a value of type `IPublishedContent`. To get the URL for that item, you need to use the `Url()` method on the item:

```
IPublishedContent result = Model.MyContentPickerProperty;
var url = result.Url()
```

Content Picker

On the page, Umbraco will render a relative Url:

`"/"`

Date/Time: The date/time picker property will allow an editor to pick a date with the option of adding a time as well. The date/time picker is useful to render data related content like a blogs published date. The code to render this picker within a view looks like this:

```
DateTime result = Model.MyDateProperty;
```

Date/Time Picker

On the page, Umbraco will render the date in this format:

`01/01/0001 00:00:00`

To format the date in a more friendly way, you can use the normal C# date-formatting techniques.

Decimal: The decimal picker property allows a content editor to add numbers with decimal places. Using its `min` and `max` fields the property can be configured to limit the value that can be added within a specific range.

```
decimal result = Model.MyDecimalProperty;
```

Decimal Picker

On the page, Umbraco will render the date in this format:

`1.0`

Dropdown: The dropdown picker property allows a content editor to select a value from a list. The property can be configured to allow editors to with pick one of more items from the list. Using the `pre value` field, you can populate the items that will be displayed within the list.

```
@foreach (string result in Model.MyDropdownProperty)
{
    @result
}
```

Dropdown Picker

On the page, Umbraco will render the data in single-option mode like this:

`string "optionOne"`

The picker will render a list in multiple choice like this:

`IEnumerable<string> ["optionOne" , "optionTwo", "optionT`

Email address: The email picker will allow a content editor to enter an email address. This property contains a regex to ensure entered emails are valid. Creating a custom email regex yourself is complicated, so this property is handy to have in your toolbox. The only configurable option exposed by the email picker is `required` which is set to `false` by default.

```
string result = Model.MyEmailProperty;
```

Email Picker

On the page, Umbraco will render an email address:

`info@jondjones.com`

Eye Dropper Color Picker: This eye dropper color picker property is new to v9 and v8.13. This picker works in a very similar fashion as the color picker. The difference is it allows for more flexible colour selection. Editor can pick a colour using HEX and RGBA rather than predefined colours. The code to render the eye drop picker looks like this:

```
string result = Model.MyEyeDropperColorProperty?.ToString();
```

Eye Dropper Color Picker

On the page, Umbraco will render the color value like this:

```
rgb(255, 0, 0)
```

File Upload: The file uploader allows editors to upload an assetm, like an image or Word document, into the CMS. Items uploaded using property will be added within the Umbraco media library. By default, these media items will be uploaded within the `wwwRoot > media` folder. As an implementor, you can then access the media uploaded onto the server so site visitors can download them. The code to get the Url for an uploaded asset is shown below:

```
string result = @Path.GetFileName(Model.MyFileUploadProperty)
```

File Upload

On the page, Umbraco will render the path to the file from the server, like this:

```
C:\myDirectory
```

Grid Layout: The grid payout property is similar to the block list editor, this property also allows content editors to add components onto a page. There are two main differentiators with the grid layout component. First, components within the grid layout are not built from document types. Second, the grid layout will allow you to define different types of layouts. For example, you could build a property with three available layouts, a full-page, a page with a left sidebar, and a page with a right sidebar. Personally, I tend to find this property less useful compared to the block list editor, however, it does have its uses. We will cover building pages with components in more detail later.

Image Cropper: The image picker and cropper property allows a content editor to upload an image into the media library. The twist with this property is that you will allow be able to define a list of available crops. For each crop you create, you can give it an alias, as well as define the crop width and crop height. When a content editor uploads an image it will also be automatically cropped using the pre-set definitions:

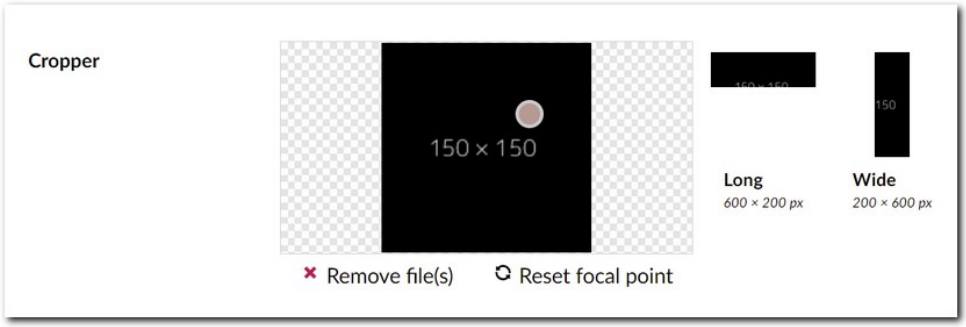


Image Cropper

The property will return the path to the image, a focal point and the available crops. To render a cropped version of the image on a webpage, you use the `GetCropUrl()` method:

```
<img src="@Url.GetCropUrl(Model.MyImageCropperProperty, "wide",
true)" alt="My Cropped Image" >
```

Image Cropper

Label: This label property can be used to allow an editor to add a label. You can restrict the type of data that a content editor can to either `string`, `int``, `decimal`, `date`, etc… The code to get the label in the HTML is shown below:

```
var result = @Model.MyLabelProperty
```

Image Cropper

On the page, depending on the restriction applied Umbraco will render the label like this:

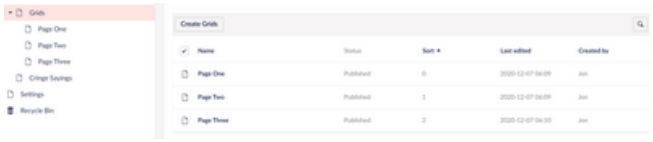
```
customText          // string
1                   // number
01/01/0001 00:00:00 // date
```

List view: The intention of the list-view property is very different compared to the other property editors. Most Umbraco property editors are aimed at allowing content to be added to a page. The purpose of the list-

view editor is to improve the editing experience easier for content editors.

Imagine having a node within the content tree that had hundreds of children underneath it. Typical scenarios where this type of build-up usually occurs are news sections and blog listing pages. Speaking for content editors, trying to find content within these types of scenarios is not fun. It is frustrating. Expanding the parent node in these troubled areas may cause unresponsiveness in the CMS. Finding the page in these scenarios is the very definition of finding a needle in a haystack.

If you have areas within your website like this, the list-view property can help. The list view property will display a list of the current page's children. You can configure the editor to render the list as either a paged-list or, a grid-view. Either view will render the data with pagination which will prevent all pages from loading at once, improving performance.



List View Property Editor

The property also comes with search capabilities, so content editors can easily filter the results. Results within the list can also be filtered in ascending or descending order to help aid in finding content. In short, if you want to make it easier for editors to find content, use this property!

Markdown Editor: The markdown editor property provides an alternative way for content editors to write content. Markdown is a very simple markup language that can be used to create formatted text on a web page. If you have ever created or contributed on a `Readme.md` file, you will be familiar with the Markdown syntax. Markdown offers an alternative way of writing rich-text content rather than having to create HTML using the WYSIWYG (WIZ-ee-wig) provided by the Rich Text Editor (RTE) property

Regardless if you use the Markdown editor or RTE when content modelling, the end output on your webpage will look the same. The main determining factor in which property you should pick this property will be based on the content editors familiarly with Markdown. If everyone on the team knows who to write content using Markdown, my preference would be to favour this property over Rich-text. Marketing focused content editors will tend to be more comfortable with using a WYSIWYG editor. Typically, only technical teams tend to be familiar with writing Markdown.

The code to get the label in the HTML is shown below:

```
var content = @Model.MyMarkdownProperty.ToHtmlString()
```

Markdown Property

Media Picker: The media picker property can be used to allow a content editor to choose one or more assets from the media library so that it can be rendered onto a page. When rendering the component, it will return list of `IPublishedContent` items, like this:

```
foreach (IPublishedContent result in Model.MyMediaPickerProperty)
{
    <img src="@result.Url" alt="@result.Value" />
}
```

Media Picker

On the page, the format of the media will be rendered like this:

`media/123/image.png`

Member Group Picker: The member group picker property allows the selection of a group. Umbraco provides membership management capabilities out-of-the-box (these capabilities will be covered later on). Members can be assigned into groups for easier management. This editor will allow a content editor to pick a member group. This property can be useful when you need to render a list of users within a group, for example top render all the users within a forum. The code to render these members is shown below:

```
string[] result = Model.MyMultiUrlPickerProperty;
```

Member Group Picker

On the page, the group alias will be returned:

`forumMembers`

Member Picker: The member picker property will allow a content editor to pick a member that is stored within the Umbraco members table. Do not confuse members with content editors. Members are external site visitors who visit your website. Content editors are referred to as users within Umbraco terminology. Users are the people who can access the CMS.

The member picker can be useful where you need to display an author's name on a page, for example, on a blog. The code to access the selected member in code is shown below. Using an object of type `IPublishedContent`, additional member information like `Author` can be retrieved:

```
var modelGroup = @Model.MyMemberGroupProperty
```

Member Picker Property

In this example, `Author` can be used to display the members name:

```
Jon Jones
```

Multi-Url Picker: This multi-Url picker property allows a content editor to be able to select multiple URLs. The URLs can be internal, external, or even links to assets within the media library. The property will then return an array of these links:

```
var modelGroup = @Model.MyMemberGroupProperty
```

Multi-Url Picker

The links will be returned in an array:

```
["www.jondjones.com", "/media/123/1.png"]
```

Multi-Node Tree-picker: The Multi-node tree propertyeditor will allow content editors to pick one or more nodes from the CMS content tree. The benefit of this editor over the URL picker is that you are restricting the editor to only pick pages from within the CMS. If you need to create links to log-in pages, or, contact us pages, this is the better editor to use.

```
foreach (IPublishedContent item in Model.MyMultiNodeTreePickerProperty)
{
    <p>
        @item.Name
    </p>
}
```

Multi-Node Picker

Nested Content: The nested content property is an advanced list picker property. The differentiator with this property compared to the repeatable text-string property is that the schema for the list items are defined using document types.

The nested content property defines 6 fields:

- Document Types
- Min Items
- Max Items
- Confirm Deletes
- Show Icons
- Hide Label

To associate list items to a nested content property, first create some document-types that are set es type 'element'. Items are associated with to the control using the `Document Type` property.

An example of how you could use this property is to deal with he slides within a carousel component. A document-type could be defined to represent a carousel item.

Another benefit of using this property is being able to generate the document-types as C# models using Umbraco ModelsBuilder. Generating models allows for a cleaner architecture.

Numeric: The numeric property control can be used by editors to enter whole a number. The numeric editor contains no customizable fields. The property works the way you would expect it to, associate it to a document type and get a number in code. The code to render the property would look like this:

```
int result = Mode.MyNumericProperty
```

Multi-Node Picker

The property will return a number:

```
3
```

Radio-button List: The radio-list picker allows a content editor to pick a value from a checkbox list. To configure this property you can add preset values into a field called `Add prevalue`. As the property returns a single string, the code to render the property would look like this:

```
string result = Model.MyRadioButtonListProperty
```

Radio Button Picker

The property then returns a string in this format:

```
optionOne
```

Repeatable Textstring: The repeatable text-string property allows an editor to create a simple list. This property is useful if you need to model an ordered/unordered HTML list. The property can be configured with a minimum and a maximum value limits. The property returns a list, so to render it in code you will need to use iteration:

```
foreach(string result in Model.MyRepeatableTextStringProperty)
{
    result
}
```

Repeatable Textstring Picker

Each item will return a string:

```
optionOne
```

Rich-Text Editor: The rich-text editor property is the classic CMS text property. The rich-text editor provides a WYSIWYG visual editor to allow content editors to create content with an ability to format it as well.

Umbraco uses a third-party component called TinyMCE as the editor within this property. This plug0in is very extendable. Within Umbraco it's possible to define what formatting options are enabled or disabled within the Toolbar. The property exposes these properties:

- Toolbar Settings

- Dimensions
- Maximum size for inserted images
- Mode (Classic or Distraction free)
- Image Upload Folder
- Ignore User Start Nodes
- Overlay Size
- Stylesheets

Slider: The slider property allows a content editor to pick a single number, or a range of two numbers from a slider. The property can be configured with 5 fields:

- Enable Range (enables a range view rather than a single value)
- Initial values
- Minimum value
- Maximum Value
- Step increments

The code required to render this property will depend on whether `Enable Range` is set or not. When `Enable Range` is set to `false` an editor will only be able to pick a single value. The code to render this value is shown below:

```
var value = Model.MySliderProperty;
```

Slider Picker

When the `range` field is `true`, the editor can add two values. These values are obtained in code using the `Minimum` and `Maximum` fields, like this:

```
var min = Model.MySliderProperty.Minimum;
var max = Model.MySliderProperty.Maximum;
```

Rendering The Slider

Tags: This tag property allows a content editor to apply a taxonomy to a document-type. This property is useful on blog and news pages where you want to provide some form of taxonomy.

```
@foreach(string result in Model.MyTagsProperty)
{
    @result;
}
```

Tag Picker

The property will return a string:

```
article
```

Textarea: This textarea property allows a content editor to add multiple lines of text. Textarea has two configurable properties:

- The maximum length of the content
- Number of rows to display

A Textarea property is accessed as as single property:

```
string result = Model.MyTextAreaProperty
```

Textarea

The property will return a string:

```
Long content
```

Textbox: The textbox property control is the standard text entry property that ships with every CMS. This property allows a content editor to add a single-line of text within the CMS. Rendering the property in code is done like this:

```
string result = Model.MyTextboxProperty
```

Textbox Picker

The property will return a string:

```
Some content
```

Toggle: The toggle property also know as the True/False editor renders a checkbox. The property can be configured with a default state (true or false).

```
string result = Model.MyToggleProperty
```

Toggle Picker

The property will return a bool:

```
true
```

User Picker: The user picker property allows a content editor to select a content editor. Do not confuse this property with the member picker. The member picker and the user picker do similar tasks, the differentiator is the data source. This property queries the `user` database table, rather than the `member` database table. This property is useful if you need to render the author on a page, say for a blog post or news article.

The code to access the selected content editor in code is shown below. The user editor will return an object of type `IPublishedContent` that represents the content editor:

```
IPublishedContent result = Model.MemberPickerProperty
string blogAuthorName - result.Author
```

User Picker

The `Author` property can be used to display the content editors name:

```
Colin Content
```

Custom Property Editors

As proven by the length of this chapter, it will be a rare circumstance when you can not model a page using CMS core functionality alone. If you uncover some unique content modelling need that you do not think Umbraco supports while undertaking a content review, it is possible you may need to create something bespoke.

The good news is that it is possible to build new properties or extend the existing ones. If you decide that you need to create a custom property, you have a lot of options. You can create new data types, property editors or extend an out-of-the-box editor using tools like a property value converter (PVC).

Umbraco will store the data added by an editor in one of four formats within the database. Using PVC, you can convert the data stored by a property editor into a different format. This mechanism allows you to create new behaviors that the original default type might not allow for. For example, `text` to JSON, or a node ID stored as an `int` to a `IPublishedContent`

You can find more about creating custom property editor from the Umbraco website here.

Document-Type Modelling Good Practices

Now you understand what content modelling tools Umbraco offers, the first step is to break your design down into components. Content modeling at a document-type level involves analyzing the designs and the content plan and determining what types of page the editors will need. On a typical project, expect to define somewhere between 5 and 25 document types. On very large projects this number might be much higher.

Taking a small brochureware site that contains a blog as an example, you might want to define several document types:

Homepage: On all projects, you will create a `homepage` document type. The homepage will typically contain all properties required to render the homepage. It will often be designed to contain additional site-wide settings as well.

Blogs: Most blogs will have a pre-set layout that looks different from other pages. A `blog` document type might be structured to allow an author name to be entered, tags, a blog image, a subtitle and a side-bar

Landing page: Landing pages are typically used to create the pages linked from within the primary navigation. Landing pages will typically have some big call to action components like a banner or a carousel at the top.

Content page: The generic content page document type is another standard CMS staple type. This document type is typically used by editors to create all other content pages, including the T&Cs and the cookie policy page

When content modelling a web page, you will also need to consider how to model the site furniture, like the header, footer, and navigation. These elements are shared between every page, so you will need to define somewhere within the CMS to allow these properties to be modelled and shared easily.

All document-types will share some common, core properties, including:

- An alias
- A URL
- A published data
- A friendly name
- A content type Id
- An icon to display in the editor
- An associated template to render the HTML

These document-type definitions are the blueprint that defines the content model that content editors can use to create pages. It defines the content contract of what's possible.

For example - It's not Home; it's Home Page. It's not a News Article; it's a News Article Page. This helps make it clear that this is a type of content that is also a webpage (and not, for example, a Data Item, like a Team Member or a Category).

When you start the content modelling process you will quickly notice that certain properties will be required on all pages. These types of properties are not covered by the default fields provided by Umbraco, however, they still need to be modelled.

SEO tags should appear in the HEAD section of every page within your site. These tags are important to ensure your web pages can be searched for and appear within search engines. The data contained within these tags tell search engines how to index your site. An example of some of these meta-data properties is shown below:

Page MetaData Tags

There are lots of different tags that you can add, the ones you will likely want to add include:

- Title
- Description
- Robots No Follow

Within your page's head section, you will also want to add the correct properties in order to model social cards. Social card meta-data can be used to allow content editors to define how your pages should appear

when linked to from social sites like Facebook and Twitter. Each social site defines their own protocols. The Twitter card specification is shown below:

```
<meta name="twitter:card" content="">
<meta name="twitter:site" content="">
<meta name="twitter:title" content="">

<!-- optional -->
<meta name="twitter:description" content="">
<meta name="twitter:image" content="">
<meta name="twitter:image:alt" content="">
```

Twitter Tags

When modelling Twitter tags, you will need to allow editors to enter:

- Site: The @username of the site
- Creator: Author name
- Title: Page title
- Description: Page description
- Image: URL of image to use in the card.
- Image Alt Text: A text description of the image

Facebook on the other hand defines the same capabilities using the OpenGraph format. This standard looks like this:

```
<meta property="og:url" content="" />
<meta property="og:type" content="" />
<meta property="og:title" content="" />
<meta property="og:description" content="" />
<meta property="og:image" content="" />
```

Facebook Tags

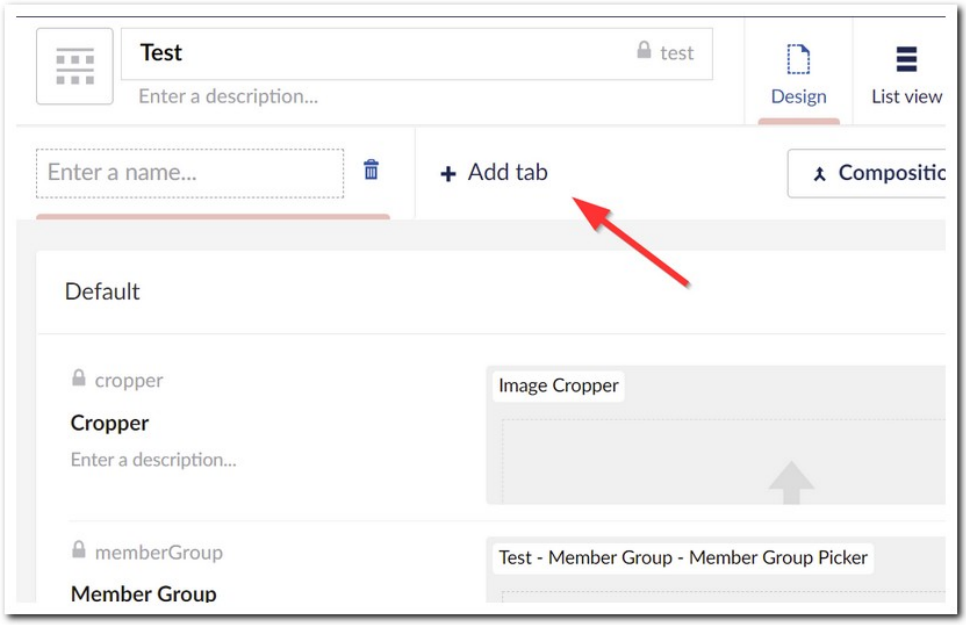
For the OpenGraph standard, you will want to model these properties:

- og:Url: The canonical URL of the page
- og:Title: Page title
- og:description: Page description
- og:image: URL of image to use in the card.
- fb:app_id: Your Insights key for analytics

Having to redefine these properties within every document-type would be very sub-optimal. Not only will the set-up of each new document-type waste a lot time, the process will be highly likely to introduce bugs.

Tabs

Umbraco also ships with the ability to organize the data types contained within your document types into tabs. I recommend you make use of tabs. Tabs make it much easier for editors to find and edit content.



Tabs For SEO Modelling

When it comes to tabs, the general content modelling advice is to model the tabs with the flow of your page. Put the most used properties on the first tab, ordered to match the flow of the page. Add less used properties, like SEO properties, Sidebar properties, etc… in new tabs. On my projects, I add all the properties related to the SEO tags within a tab called `SEO`. Not very imaginative, however, very easy for content editors to understand!

Using tabs to separate properties into smaller grouped chunks, will also aid in understandability. Not overwhelming the content editor with lots of choices will lead to a nicer editor experience.

Sharing Settings

The next challenge to solve is where to put the SEO tags? Duplicating these properties onto every document-type is sub-optimal, so what are the alternatives?

Umbraco provides two ways for settings to be shared between document-types:

Inheritance: When you visit the `Document Types` node from the 'Settings', you will hopefully notice that the document types are laid out in a tree structure. This is because Umbraco supports document type inheritance. Umbraco provides this capability by allowing document types to be nested on creation. Using the classic parent/child relationship mapping, it is possible for child document-types to inherit properties defined in the parent.

If a document contains some properties and then a document-type is created underneath it, that child will have access to all the same properties as the parent.

There is something you should be aware of around document-type inheritance, it's not possible to change this relationship after its been made. When a document-type is created it is not possible to move it later. If you want to use inheritance for SEO, you need to build this nesting in from the start.

Deleting and re-creating a document type is the only way to add or remove inheritance after creation. At a later date, if the inheriting properties are no longer useful, you will need to remove that type and any content generated using it. If the site contains a lot of content that has been created using that type, this refactoring might be deemed too much effort.

Due to this limitation, I recommend that your first task should be to create a `base` document type that all document types within your project used to generate web pages inherit from.

I always favour document type inheritance to create a shared SEO tab between all pages. The process of setting this up is simple. Create a new document type and set the alias to `BasePage` or something similar. On the editing page, create a tab called `SEO` and add the relevant meta-date properties. Finally, create all other web page related document types underneath `BasePage`.

Composition: Document types created with the `Composition` flag enable allow you to structure a document-type on a component-level. Components are document types that form bits of pages, however, they can also be used to model groups of settings that can be applied to a page.

Composition that items can be referenced by other document types. For the SEO example, you could create a new document-type and set its type as a composition. Within this SEO composition document-type, you could create a tab called `SEO` and add the relevant fields:

Composition Document Types

To associate a composition onto a document-type, click on the big `Composition` button from the documents types edit screen. This will allow you to select the document type that you would like to associate. As seen in the screenshot above, the settings defined in the SEO composition will now be inherited.

The downside of using compositions for every page is that you will need to associate it each time you create a new document-type. With inheritance you simply need to create a document-type underneath an existing one.

The benefit of this approach is better flexibility. It is much easier to change properties and add different things using compositions. My general rule, is SEO meta data wil never change. Adding these properties using inheritance is OK. For modelling any other type of shared settings that I might requires, I tend to use composition.

When creating new document-types within the CMS it is possible to organize types within folders. I recommend at the start of a project creating three folders, one called `DocumentTypes`, one called `Compositions` and one called `Elements`.

Hybrid: The final option is to use a hybrid approach. In this pattern you mix inheritance with composition. To implement a hybrid pattern, you could create a blank base document type that all document types inherit from. The SEO data could be modelled within a new SEO composition document type. The SEO composition could then be applied onto this base type. After a composition has been attached with a base document type, all child types will automatically inherit all the properties defined within that composition.

This mix and match combination provides the best of both worlds. Having a base type will give you the option to easily apply global properties onto all pages now and in the future. Using compositions will make it easier to add, remove or completely change groups of settings. In v9 and upwards, this is the technique that I recommend you adopt.

Managing Global Settings

The focus of the content modelling process so far has been on a per-page basis. When presented with a set of wire-frames, you will often come across components, content and settings that are shared between many pages. The website's header, footer, and primary navigation are all classic examples of these types of components.

Assuming each shared component will need to have associated properties within the CMS to allow content editors to update the website. Where should these properties be added?

Out-of-the-box, Umbraco does not provide a specific mechanism for managing shared components or global settings. It is left to the implementor to decide how they would like to model these things within the CMS. Um-

braco CMS has been used to build websites for over 20 years, so obviously there is a solution. The challenge is picking the one that works for you!

As Umbraco does not ship with settings management capabilities, you will need to model these shared properties somewhere in the content tree. This causes an interesting content modelling juxtaposition. The content trees' main purpose is to model pages on a per-item basis but we also need to squeeze in these extra settings inside of it as well. Just so we are clear on what you might need to model, some common examples of these types of global settings could include:

- API access tokens that need to be updated by an editor
- Site title
- Social media IDs
- Links to key pages within the site
- Footer content
- Header content
- Content required by a mega-menu
- Email templates

In this section, you will learn about all of the known patterns that Umbraco developers use to solve this dilemma. Defining which solution out of this list is the 'best' is very subjective. My recommendation when selecting which technique to use is to ask the team. Present the options to the team who will be updating the content and get their feedback about their preferences. This is a content modelling 101 rule, asking the team is better than assuming. If asking the team is not possible I will also highlight the pattern that I use. Pick that one.

In terms of content modelling, there is no one universal rule that can be applied to every project. All the patterns listed here have been successfully used in production and will work, so pick the route that makes you happiest.

Settings Composition

It is possible to create document types as compositions. Document types that have been created as compositions will not be treated as an individual page. Compositions are treated as components that can be attached to a normal; document type.

One way to deal with global settings is to create a settings document-type composition. When creating a composition to model shared components you have two options. Create one composition and model all the properties in it, or, create multiple compositions on a per-component basis.

In terms of good software engineering practices, it is common knowledge that breaking things into smaller components is considered a better practice. This rule applies to class design, method design, service design and

content modelling. Creating a composition per things component will your solution easier to update and maintain moving forwards.

Following this pattern, I tend to create a folder within the `Document Type` node within the `Settings` section called `Settings` or `Shared`. In this folder, I create a one-to-one mapping of components to composition files.

Modelling the properties required to render a shared component is exactly the same process required to model a page. You will be presented with the same screens, property editors and data types. The only difference is these compositions will need to be associated with a page, rather than accessed directly.

Settings-Block Pattern

Rather than applying settings directly onto a page, an alternative approach would be to model global properties within a block and then associate that block onto a document type via a property editor.

To apply the settings-block pattern, you would group related settings into a block. For example, you could create a header settings block, or, a footer settings block. To define a block, create a new document type and then set that type as an element.

A document type that has been configured to run in `element` mode, can be configured in exactly the same way as any other document type. Element document types can be used as blocks within the block-list editor.

The nice thing about using the block-list-editor and blocks to model global data is that it provides a nice layer of abstraction. The benefit of this abstraction is easier site and settings maintenance. In the future, if the site's header needed to be redesigned, a brand new menu settings block could be created and applied.

Creating a brand new block to model new changes reduces deployment risk. If you need to refactor existing settings the initial deployment is very risky. What happens if the new change does not work? There is no easy fallback process except to revert the database.

When using blocks to model settings, it would be possible to decouple a release from a deployment. During a big site update, it would be possible to push new changes live and turned off. This on/off check could be done by checking for the existence of the related settings block.

After the deployment, an editor could create the associated settings block. Creating this block would then enable that feature on the production website. Once activated the new feature could be tested by a QA. If the QA fails the new changes, the newly created settings block can be deleted to automatically disable the feature on the website. When testing is eventually signed-off, the old settings block that has been replaced can be deleted.

If you decide to use compositions or blocks to model settings, you will still need to decide which page to add them onto. The decision over which page to use segues nicely into the other two popular patterns developers use to model global data.

Ultimately, the only way to add content inside of Umbraco out-of-the-box is on a page. You can make your life easier by modelling settings into a composition or a block, however, you will always need to decide on the page that those settings will be attached onto within the CMS.

The approach used in the remaining patterns involves modelling global data directly onto a specific document type. This means that regardless of which pattern you use, you still need to make a decision about which is the most appropriate document type to use!

It is definitely possible to create a custom solution to deal with global data. There is nothing stopping you from creating a custom backend screen, creating a new database (SQL or NoSQL), and then writing all the CRUD code to wire it up. All these tasks will obviously involve custom development work. This code will also need to be maintained and performance tested. Creating a custom solution is not a trivial task. I have yet to justify the effort of creating a custom solution, so on my projects, I always model global properties onto document type.

Most teams will typically decide to use either the existing homepage document type, or, they will create a new settings specific document type to model global data.

The Homepage Settings Pattern

The first benefit of selecting the homepage as the central store for global data is that you do not need to create any additional document types. Every project needs a homepage document type and all pages within a website will be created underneath the homepage.

Another reason why the homepage settings pattern is so popular in CMS development is that every CMS vendor provides an API that allows easy access to the homepage. Umbraco is no different. The content API will be covered in the next chapter, for a quick reference the code to access the homepage is a simple one-liner

```
public ExampleCode(IUmbracoContextAccessor context)
{
    var homepage = context.Content.GetByRoute("/");
}
```

Get Homepage In Code

The main disadvantage of adding lots of properties onto the homepage is data bloat. On smaller websites, this bloat may never occur, however, when the size and the complexity of a site grows, typically the number of global data that need to be modelled also skyrockets. In terms of performance, the more data the API needs to retrieve the longer it will take. Typically, the homepage will be the most popular page on a site, so potentially making this the least performant page is not ideal.

Another issue with this pattern is maintenance. The homepage is typically the most visited page within a client's digital estate. Bugs on the homepage will consequently impact the widest audience.

Over time once used properties may become defunct. When your website contains unused properties, it becomes harder to understand. To ensure good CMS hygiene, expect that you will need to refactor, edit or remove existing properties once in a while.

Refactoring existing properties within any production website will involve some level of risk. Adding global data on the homepage simply increases the chances of you introducing a big impact bug.

Imagine removing a setting that you think is obsolete. Unbeknownst to you, this particular setting is used on the menu. You refactor the CMS and delete the setting. This causes the code in the menu to throw an exception. With the sites header throwing an exception, every page on the site will now be broken. I have seen this exact scenario occur several times during my career. While unlikely to happen, it is something you should consider before making an architecture based decision.

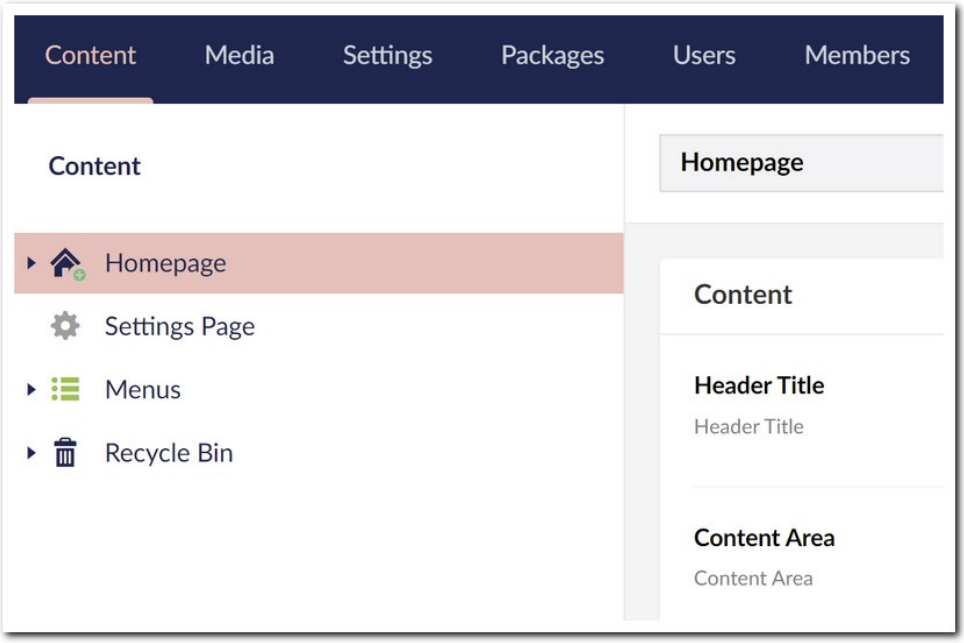
Finally and potentially most importantly, you need to consider governance. Some of the global settings will be essential in order for the site to function. If a content editor misconfigured a site critical setting, they could accidentally take the whole site down. Most content editors will need the correct permissions to make content amends to the homepage. When the homepage document type contains global settings and content properties, there is no way to define which content editors can access and update settings and which ones can update the homepage content.

Settings Page Pattern

The alternative approach is to create a specific settings document type. In this pattern, all global data can be modelled in the main tab from the settings page. This means one less click for editors!

Within the content tree, the settings page should be created outside of the main website hierarchy. The settings page should be created as a child of the root node rather than the homepage. There is a security benefit from locating the page here. It is likely that you will need model sensitive data like access tokens and API keys. Modelling sensitive and security-critical properties onto a public-facing document type is not ideal. Adding properties onto a page that is not accessible via a URL has security benefits.

If you want to apply this pattern within your project, the structure of the content tree would look like this:



Settings Pattern

I personally think that using a specific settings document-type definitely has a lot more advantages compared to the alternative. To recap. these advantages include:

- A simplified homepage. When the settings are modelled on a settings page, the homepage will be less cluttered, making it easier for content editors to work with.
- Better governance. You can apply a more fine-grained security policy over who can access and manage settings.
- Settings are easier to maintain in the future because it is less risky to refactor them
- No performance impact on the homepage
- Sensitive settings are not stored on a public-facing document-type

The main disadvantage of using this pattern include:

- An additional document type needs to be created
- The code to call the settings page is one line longer!

Out of these approaches, I tend to favour using a settings document-type combined with one or more settings blocks within my projects. I have found that this pattern results in less risky deployments. It also helps combat future technical debt. Set-up takes a few extra minutes and there are no big disadvantages of modelling settings this way.

Block Modelling

During the content modelling process, it is likely that you will identify components (more commonly referred to as blocks) contained within the designs that are reused in multiple places. When you identify a shared component, aim to avoid re-modelling. Instead, I recommend you focus on re-use.

Block modelling is considered a good content modelling process as it results in a much more modular design. Block modelling is a practice that you should definitely aim to incorporate within your project. When content editors have the power to structure a page layout themselves, they have more creative freedom. The benefit of this content flexibility for the development team is a reduction in support requests.

As time progresses, every company's strategy will eventually change. Campaigns come and go. Customers' needs and expectations will also change. The layout that converts well today, might not be as optimal in six months. If you design Umbraco so content editors can deal with these inevitable changes, they will not need as much developer help!

Every project and design that you work on, will contain features that can be modelled as blocks. These blocks might include call-to-action blocks, image blocks, carousels, banner blocks, and embedded video blocks.

Blocks will also help define a consistent system-wide experience. For example, a typical banner could be composed of an image picker, a title, and some supporting text. Without using a block, the only way to model content shared across different pages is to re-create the fields on all of the associated document types. This duplication of fields will likely lead to small backend inconsistencies. The naming of a field or its description might slightly differ. One might be called `Title` while another `Banner Title`. Even though designs for the banner may look the same on the frontend, the backend content modelling experience can end up being very different.

Some readers might think these nuances are unimportant, however, the devil is in the detail. In order to build a great platform, creating a uniform experience is important. Duplication will also mean that you will need to write double the amount of code and HTML, so there is an added developer time-saving bonus for using blocks.

The last benefit of using a more block centric design is an often overlooked subject, personalization. Out-of-the-box, Umbraco is not a strong CMS contender for personalization. This is not surprising as Umbraco is free and open-source. All the great personalization services are aimed at the enterprise. Enterprise tools have AI and other fancy features. Subsequently, they all cost a lot of money! Even though Umbraco is not the strongest in terms of personalization, it is definitely possible to create personalized experiences in a Umbraco powered website.

By combining the tag property with feature toggles in code, you can decide which blocks can be rendered to which audience. This type of on/off capability is only possible at the component level and not on a page level.

Umbraco ships with several property editors that allow for block modelling. Each property editor behaves slightly differently, however, ultimately they all allow you to define an area on a page where content can be created based on a pre-configured block library.

To start block modelling you will first need to identify a list of blocks from the designs. Next, on a per document type basis, you need to define an area where blocks can be added. For each individual area, you need to pick the most appropriate property editor.

How To Model A Block

In the last section, we took a high-level look at all the property editors in order to understand the content modelling capabilities available to us. In this section, we will solely focus on the editors that allow for block modelling, being:

- Block List Editor
- Nested Content
- Grid Layout

Out of these properties, the block editor is the latest and in my opinion the most useful.

Block List Editor: The block-list editor was released in Umbraco v8.9, which means it is the new kid on the block. The block-list editor definitely fills a much-needed content modelling void and personally, I think it is one of the most powerful properties within your Umbraco content modelling toolkit.

Historically, a lot of community packages, like the defunct Doc-Type Grid Editor, Bento Editor, and Perplex Content Blocks were created to fill this missing content modelling need. Now we have this capability and there is no need to install any third-party packages!

The great thing about this property is that it allows for blocks to be defined using a document-type. From a code perspective this is very useful as it allows for model-first development.

Any block based document-type can be used in conjunction with the Umbraco Models Builder. Models builder can be used to generate C# classes that represent each block. Using C# classes to interact with the CMS, will give you all the power of strongly-typed models, Intellisense and all that good stuff.

There is one aspect of the block list editor that makes me sad. When it comes to rendering blocks, out-of-the-box the property will pass the data directly into a view and not via a controller.

By default, the property does not provide any from of block-level route-hijacking. As a result, you will likely need to write code in your views.

If you believe in the power of the MVC paradigm, code in views is not ideal. The good news is that by overriding the default block list editor view, it is possible to change this behavior. It is possible to update the code in this default view to render a block via a controller rather than just call a view directly. How to perform this tweak will be covered in the rendering component chapter later on.

When it comes to block-modelling, the block list editor will likely be your go-to property. As the block creation process uses document types, you will not need to learn anything new. There are no limits as to what property editors you can use inside a block, so you can build simple and very complex blocks easily.

When you need to model a block selection, the block-list editor is the property that I recommend you use unless you have a good reason not to. There are alternative properties that can be used to model components and in certain scenarios, these alternatives might make more sense. We will cover these scenarios next.

Grid Layout: The gird layout released in v7.2, was the original property that Umbraco released that provided some form of block creation ability. The differentiator with the grid layout property is layout selection. The grid layout property allows you to define an area on a page where a content editor can then subsequently build a virtual grid. Content editors can define the grid shape using templates that are defined as part of the data type contract. These templates allow an editor to select the number of columns and their alignment. After picking the number of columns, the editor can then select the type of row they would like to create inside of each column. Editors can then add predetermine blocks inside of the cells within each row.

Like all things Umbraco, the property can be configured and customized to your needs. Using the out-of-the-box configurations, when a content editor is presented with this property, they will first be asked to select a grid 'layout' with the default options. The out of the box templates are called '1 column layout' and '2 column layout'.

After choosing a layout style, an editor can then add rows into the columns. The layout of these rows is also configurable. By default, you get a full-width row called `Headline` and a two-column row called 'Article'.

After picking a row, an editor can then add one or more blocks inside of the cell. Out-of-the-box, the available blocks are Embed, Headline, Image, Macro, Quote and Rich-text editor.

The gird layout exposes these properties when you add it onto a document type:

- Grid Layout
- Row Configuration
- Number Of Columns

- Settings

As the property can be populated with blocks of type Rich Text Editor, it also allows you to configure RTA from the creation screen. These properties are exactly yhe same, so you you have options to define the RTE toolbar icons, the stylesheets, mode, etc… Adding the define which rows and columns are allowed for that instance. To render the grid within a view, the `GetGridHtml()` helper can be used. `GetGridHtml()` takes two augments. The data model and the alias of the grid property:

```
@Html.GetGridHtml(Model, alias)
```

As the Umbraco documentation confirms, the grid layout property is not the optimal property to use in every block modelling scenario.

The great thing about this property is the flexibility it gives to editors for content production. If the designs/wireframes define an area where content needs to be created in a grid, however, that flow still needs to work within the boundaries of the overall site, the grid layout property is a good option. If you need to model components that are more complex than just simple bits of content, be warned you will hit two limits.

The first limitation is due to the editors rendering behavior. It is not possible to change or customise how blocks are rendered within this property. This presents a big problem if any of the blocks need to render content that is stored externally to the property.

If any of the data required by a block is stored within a SQL database, a settings page, or even a third-party system, there is no way to incept the block rendering process to get this data. You will not be able to associate a controller with a block.

There is a workaround to overcome this limit, however, it is not ideal. Blocks within this property are still rendered using a view. This means it is always possible to write custom code directly within the view to do whatever you need. Writing code in a view is very hacky. You can not unit test that logic. It is also less performant.

The other major limitation of the grid layout property is the content modelling capabilities offered. If you need to build blocks within the grid that only render text, images, numbers, or URLs then life is easy. If you want to use properties like the block list editor, list view, etc… within your blocks, you may struggle.

If you inspected the types of properties used by the default grid editor blocks, you will notice they are all simple types. The majority of these default blocks are constructed from a single property. On your project, if that is all you need to model, this property works really well.

If your blocks need to be more complicated use the block list editor. In the grid view property, blocks are defined in code and config rather. This means that you can not organize properties into tabs or sections like you could within a document type powered block. You will also need to do a code deployment to make changes to the blocks!

I find it helps to conceptually consider the grid layout property as an advanced alternative to the rich text editor (RTE) property. This property is great for providing editors with the ability to have a lot of control over how simple content should be added onto a page.

The grid editor is not the easiest property to use and it will take more effort to set it up compared to RTE, however, it allows content editors to be very creative.

Nested Content

The nested content property will allow a content editor to build a list of items based on a schema defined by a document type. Admittedly, the nested content property is not a pure block focused property, however, it is definitely handy when you want to define a configurable area on a page and you want to have more control about what content can be created inside of it. The nested content property is definitely useful if you need to model something like a data table in the UI. Providing editors with the ability to build lists based on complex items (with validations if needed) allows you to model trickier visual components and have confidence that an editor will not accidentally break the styling.

Granted there is not a huge difference between the capabilities that this property provides compared to the block list editor. The block list editor is definitely a lot more powerful as it provides more options to customise how blocks are rendered. The block list editor is a better choice when you need to model blocks that contain lots of properties.

The reason for this is that data entry within the block list editor property is performed within a side panel. Using a side panel allows for Umbraco's infinite editing abilities to be used. Sections, tabs and even nested block areas defined within the element can be respected and navigated easily. The only minor downside of the data entry being made within a side panel is that the editor is jumped slightly every time they create a new item in the list.

Within the nested content property, item editing is done in a list on the page. There is no jumping. No additional tabs, etc… If you simply need to display a list, use the nested content property!

To create an associated list item schema, go to the `Document Type` node within the backend, found within the `Settings` section. From here, create a new `Element` document type and define the fields that you want to be available on a per line item basis. As you have previously learnt, `Element` types are used as components rather than pages!

When applying the nested content property onto a document type, you will be able to configure the property using these parameters:

Element Types: Defines the document type that will be used as the list-item schema. For each item you add you can define three properties:

- Element Type: The document-type to use

- Group Template: You only need to set this property if you want the item schema to be based on properties not located within the first tab from that document type.
- Template: The associated view that contains the HTML for the item

Label: Defines the label used to render the list items. You can use the `{{propertyAlias}}` modifier to access the default value

Min Items: Defines a validation rule that specifies the minimum number of items that need to be created

Max Items: Defines a validation rule on the maximum number of items that can be created

Confirm Deletes: When enabled, editors will be prompted to confirm they want to delete items first.

Hide Label: Renders the property in full-width mode and hides the label!

There is one caveat about relying on nested content too much. Rumours are floating around online, that the property will eventually be deprecated in some future version of Umbraco. As you can model the majority of the same functionality using the block list editor alone, why have two very similar properties?

If this rumour is true, the impact it has on your project is some potential re-work in the future. At some point when you upgrade, you may potentially need to remodel some content! As the whys and whens are still very uncertain, I would not let this tidbit of information influence your content modelling decision-making process in the here and now!

Content Modelling Best Practices

The final topic to discuss in this chapter is best practices. During the content modelling process, keeping some fundamental practices in mind and incorporating them into your builds will help take your builds to the next level.

The overarching principle for strong content modelling design is precision. No matter if you are writing code, content, or, content modelling, the more precise you can be, the more clear and straightforward the resulting output will be. The more precise that you can be when content modelling, the less ambiguous the system will be for editors to understand. In order to build anything of high quality, details matter. As the old saying goes, the devil is in the detail. This is certainly the case for good backend CMS design. There are certain areas where it definitely pays to be precise, these areas include:

Naming Conventions: It is considered good practice to always use clear and descriptive names when creating a new document type. A good naming convention should be descriptive and consistent.

For example, as you have learnt, not all document types will be used to create public- facing pages. The purpose of some document types, might be to store settings or a component. For this reason, one very trivial but often overlooked convention is to always include the word `Page` or `Page Template` in the name for all document types that will be public-facing.

Let us say you need to model document types for a settings page, a menu container and a home page. The quickest and laziest way to name these document types would be `Home`, `Settings`, and `Menu`. Names without a description are slightly less meaningful compared to more descriptive names like `Home Page Template`, `Global Settings` and `Menu Container`.

Avoid Acronyms: When deciding upon a name, try to avoid using acronyms. An acronym might make perfect sense to you, however, this might not be true for every editor using the system. For example, the intention of this label, "Enter Vehicle Identification Number" is a lot more obvious and intuitive, compared to "Enter VIN".

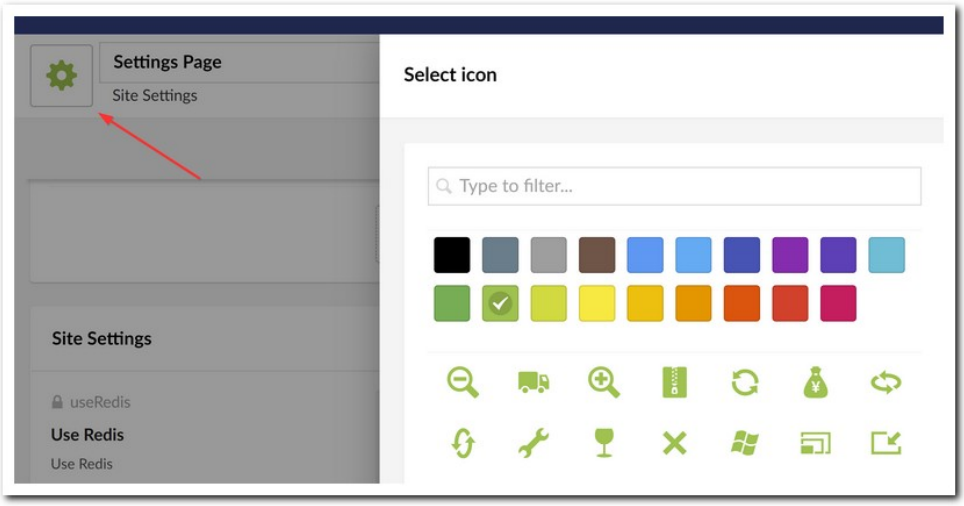
Spelling: We are all human and when content modelling, spelling mistakes and typos can happen. To reduce typos, I find it helps to define a process to catch spelling errors.

Some options to improve spelling might be to install a browser-based spelling tool like Grammarly. An alternative would be to write all content in MSWord first, run a spell check, before copying the content into Umbraco.

Icons: When setting up a new document type you can associate a custom icon onto it. The icon will be shown to the left of any page created using that type within the main content tree. Custom icons make it visually easier for content editors to differentiate content.

Umbraco ships with a fairly large icon library out-of-the-box that you can select an icon from. You do not need to upload custom icons into Umbraco. Umbraco ships with icons for settings, content, media, members and more.

Assigning a more appropriate icon to a document type will make it that little bit easier for an editor to quickly identify content within the CMS. Custom icons provide an additional visual feedback loop about the site's content structure and I definitely recommend you make use of them!



Umbraco Icons

You can associate an icon to a document type from the document type editing screen. Click on the blank icon to the left of the document types name/alias field. This will launch the icon picker dialog. From this screen, you can pick the icon that you would like to associate with that type. You can also assign an icon colour.

Colour grouping is another small tweak that can help relate similar types of content. For example, I tend to add all setting related document types with a grey icon.

Limit Choice: Overchoice is a cognitive impairment in which people have a difficult time making a decision when faced with many options. When creating new document types makes use of 'Allowed child node types' and 'Allow as root`from the document types`Permission` tab.

Being very strict and limiting what document types content editors can use in what area will not only make the CMS easier to use, it will also help eliminate bugs. It is these small touches that will make the CMS that extra bit nicer to use.

Validation: When adding properties onto a document type you should always aim to add validations and conditions. Should this bit of content be mandatory, can the sub-title be optional? In the haste to set-up a CMS it can be tempting to forget to add these restrictions, however, defining these rules can prevent an editor from accidentally breaking the pages layout. Most of the validation advice around content modelling is pretty obvious. The same rules apply here as they do when creating a web form:

- Should the property be required or optional?
- Should this text have a max length?
- Should this number be limited to within a range?
- Do we need time and date?
- Does this field require advanced validation, e.g. a regular expression?

- Is the validation too strict? There are no universal rules for validating fields like name, address and email. It's equally as important not to be too specific on formats or characters when validating. If the validations are too strict, the content editor will not be able to use the system!
- Does the error message make sense? If you need to provide an error message, use positive and helpful language

Flow: To optimally structure a document types so that it is easy and efficent to use by content editors, consider the flow. In general, the flow should match the elements as they appear on the web page. Put the most used and mandatory fields at the top of the structure. Put the optional and less used fields at the bottom

You should consider which elements that you want to be grouped logically together. Add these related groups of properties into tabs or sections.

Folders It is possible to organize the document types into folders within the doc-type node in `Settings`. Organizing these types into folders can make it that little bit easier to locate things within Umbraco. The grouping rules apply equally as well here. You could create folders for `Pages`, `Components`, `Settings`, `Containers`, or `Elements`.

The output of the content modelling process is a bunch of schemas that will define all the different kinds of content that can be created within your website. These models will define the foundation for how your site is built. I advise that you give yourself enough time to spec everything out.

Usually, the hardest part of content modelling for Umbraco newbies is knowing how to map a design to a Umbraco component. Breaking content types down into the most optimal component can be frustrating.

You have lots of content creation options with Umbraco out-of-the-box. As long as you follow the rules in this section, feedback your thoughts to the content editing teams so they understand how and where their content is used. The content modelling process will usually go pretty smoothly.

How To Build Pages Within Umbraco

After completing the content modelling process, you will have a set of document types. Each one of these document types will represent a page that can appear on the website. Your next job will be to create all the corresponding code to render that page. This code will need to take the data entered by the content editor and convert that into beautiful HTML.

In this chapter, you will learn about the overall mechanism behind this rendering process. You will learn about the components and mechanisms that Umbraco provides in order to aid in this creation, as well as the theory of how it all hangs together.

There is good news for the Umbraco old-timers is that the process to render pages has not changed between versions. All the controller types are the same as in v8. There are a few nuances within .NET Core, mainly around the return types, however, overall it will feel very similar.

Under the hood, you will still follow the principles of the MVC framework. Applying route-hijacking to map a document type to a controller, performing any business logic within that controller and passing that data down via a view model. Finally, using that data to render HTML within a view.

One slight but very important difference in Umbraco v9 is the location where routing rules are applied. Historically, the default routing rules were defined within `UmbracoApplication` and custom routing rules could be added within `global.asax`. Within .NET 5 both of these types are deprecated and no longer in existence!

For Umbraco newbies, I highly recommend that you invest the time to understand how CMS routing works. Understanding how the routing pipeline works will pay dividends.

Whenever I need to learn a new CMS, I find understanding the exact steps a request needs to follow through the rendering lifecycle is a fundamental topic. When you can fully comprehend the path a request follows, it helps clarify why a CMS has been built a certain way.

The default CMS routing logic is fundamentally different compared to a vanilla .NET Core website. This is why the topic of routing rules is the first section in this chapter. Understanding the whens and the whys around routing rules should make everything else within this chapter click into place that much easier.

Routing And Route Hijacking

When you start to review the output from a content modelling session, it is very likely that you will have identified certain pages and features on the website that have differing rendering needs.

Most requests will be triggered from the website, to get content created within Umbraco and render it. Some requests might require the processing of data that has been posted back from a form contained on a web page. You may even need to handle a request that Umbraco cannot support by default, like a request to a vanilla MVC controller.

Not all requests will be via the website. Some requests might come from a custom backend screen you need to build, others might be to a custom API. The main takeaway is that during a CMS build you will encounter different types of requests. For each type of request, the routing rules will vary.

Out-of-the-box, Umbraco ships with six different controllers to help you deal with different types of requests. You can also still use the classic and much-loved vanilla .NET `Controller`. The majority of the Umbraco controllers will work out-of-the-box without you having to do anything, some will not. The difference… the routing rules.

Without understanding how the pipeline works, whenever you set up a new element it is likely that you will encounter errors. I have wasted hours of my life wondering why the website is throwing a 404 error rather than rendering the expected content. Getting the routing to work can be very frustrating and if you do not understand how CMS routing works, it will feel like you are swimming in treacle.

The default CMS routing rule differs to the default vanilla MVC rule. This default MVC routing rule is shown below:

Default Routing Rule

This rule is telling .NET to match any URL, to a controller and an action. If no additional segments have been supplied within the URL, e.g. a request has been made to the homepage (`\`), then attempt to load an action method called `index` inside of a controller called `home`.

If a request to `www.website.com` was made, this would trigger the rule and the `home` controller would be loaded. In vanilla .NET, the segments

within a URL need to map to a controller and an action. As long as the correct code exists within a solution, the routing just works.

In vanilla MVC, you can simply create controllers and in most instances, the default routing rule will connect the request correctly and auto-magically.

Within a Umbraco powered website, this default rule does not work. Umbraco needs to override this default vanilla routing rule.

Umbraco Routing Rule

The reason why Umbraco needs to use a different default routing rule is related to data. Inside a Umbraco powered website, when a page request is made there is no direct correlation between the segments in the URL and the controller and action that will be used to render that request.

It would be highly impractical and absurd if a developer needed to write a controller per page that was created inside the CMS. Instead, within a CMS we need a way to create a single controller that can deal with multiple pages.

Within Umbraco, pages are defined within the CMS. This data ultimately lives inside of the Umbraco database. The segments within a requesting URL need to map to pages that have been created within the content tree.

For each segment within the requesting URL, a check needs to be made that a corresponding page exists within the CMS. If a match is found, the next segment can be processed. If not, throw a 404 error. This is why the routing rule logic within a Umbraco project differs.

Within a Umbraco project, the main commonality between different pages is the document type that it was built from. To avoid the need for a developer from having to create a controller on a per URL basis, in a Umbraco architecture requests are mapped based on the document type used to build that page.

When the routing works at a document type level, a controller can be created that can render multiple pages. This allows for greater code reuse and makes CMS development possible.

Within a Umbraco request, an attempt is made to map the incoming URL to a content page created within the CMS. If a match is made, Umbraco will determine the document type used to generate that page. It will then try to call an associated controller based on that document type. Unlike vanilla MVC, the request does not simply map to a controller based on the URL.

During a successful look-up process, Umbraco will also add all the page related data into the request context before re-directing the request to a controller. You can then access this page related data within the controller, do what you need with it and then eventually render the final HTML.

The key takeaway is that within a Umbraco project, the normal URL to controller/action mapping will not work. Out-of-the-box, the routing within a Umbraco website is configured to allow for Umbraco pages to be rendered. If you want to do anything different, you will need to manually define a custom routing rule and then add it within the applications route table.

Umbraco provides 6 different types of MVC controllers. When you use the controllers that are related to rendering pages, you will not need to think about routing rules and everything will work.

If you want to create a backend screen, an API, or you simply want to reserve a certain URL and call a normal vanilla MVC controller, you will need to create a rule.

The difference in routing, is one of the main reasons why understanding how the page pipeline works is important. If you attempt to use a controller type that will only work if an associated routing rule is defined, hello 404!

If you want to map a specific URL like `www.website.com/hello` to a controller, you will need to define a custom rule. This process is the exact opposite of how you would normally build a website using .NET!

Shortly when you learn about the different controller types, always keep routing in mind. The good news is that Umbraco will handle most of the routing for you. A good general bit of advice is that whenever you encounter a 404 when setting up a new controller, have a look within the routeing table!

Adding Custom Routing Rules

Before we delve into the different types of controllers, you will need to understand how to add custom rules. Without understanding how to implement a custom routing rule, you will not be able to get half of the controllers to do anything meaningful!

Within the .NET Framework, developers could add custom routing rules within `global.ascx`. As `global.ascx` no longer exists, the process for adding these rules within Umbraco v9 is different compared to adding rules within v8.

Within a .NET Core powered website, custom routing rules can be added within `Start.cs` within the `Configure()` method, although as a best practice, I do not recommend adding the rules here. I think a better coding practice is to create a custom routing middleware extension (which will be covered shortly).

If you look within your solutions `Configure()` method, you should see the code that registers the Umbraco middleware and its endpoints:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseUmbraco()
        .WithEndpoints(u =>
        {
        });
}
```

UseUmbraco()

For brevity, I have simplified the code snippet above, so expect to see more code in your `Configure()` method. You can add custom routing rules directly within `.WithEndpoints()`, using `EndpointRouteBuilder.MapControllerRoute()`.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseUmbraco()
        .WithEndpoints(u =>
        {
            u..EndpointRouteBuilder.MapControllerRoute(
                        "custom-route",
                        "example",
                        new { Controller = "MyExample", Action = "Index" });

        });
}
```

UseUmbraco() With A Custom Rule

In the example above, this rule above would redirect any requests to `www.website.com/example` and forward them onto a action method called `Index` within a controller called `MyController`. To create a new rule you will need to define four properties, two mandatory, two being optional:

Name: This is an internal name used to identify the rule so you can use whatever name pleases you.

Pattern: The pattern that will trigger a matching

Defaults: Allows you to defines any default/fallback values. It is handy to always include the default `Index` action method here. This result in smaller and nicer Urls within your application. Another common default is setting `id` as `UrlParameter.Optional`

Constraints: Constraints are new to .NET Core. Constraints can be used to strip unwanted data from reaching the controller

In a typical project, you may need to add ten, twenty or even more custom routing rules. Adding a handful of routing rules within `Start` is fine, however, not ideal. The reason for this boils down to responsibilities.

`Start` is the main configuration class within a .NET Core application. Depending on what your website needs to do, will determine what configurations you will need to add within `Start`. Typical configurations will include dependency injection, middleware registration, cookie/CORs/security policies, logging configuration, redirect rules, and more. The important takeaway is that the `Start` class is very important and it can contain a lot of code.

The single-responsibility principle (SRP) is a best-practice bit of advice that states “a class should have only one reason to change”. The reasoning behind the single-responsibility principle is that the more responsibilities a class has, the more you will need to update it.

When you have to update a class that is responsible for lots of things just to change one thing, you risk accidentally breaking something completely different within that class. Just like a game of whack-a-mole, you make a change to fix one bug and the next thing you know another occurs. The more responsibilities you give `Start`, the higher the odds something else will break when you need to change it.

To apply the single-responsibility principle to `Start`, it is a better practice to encapsulate all your custom routing rules within a separate class that is solely dedicated to routing. This architectural approach will reduce the chance of you accidentally breaking unrelated routing configuration code within `Start`.

The code to create a dedicated routing class is simple enough. You will need to create an application builder extension class and then call it from `Start`. The code to create the builder is shown below:

```
public static partial class UmbracoApplicationBuilderExtensions
{
    public static IUmbracoEndpointBuilderContext UseCustomRoutingRules(this
IUmbracoEndpointBuilderContext app)
    {
        app.EndpointRouteBuilder.MapControllerRoute(
                        "AdminDefault",
                        "umbraco/backoffice/plugins/<CONTROLLER>/<ACTION>",
                        new { Controller = "<CONTROLLER>", Action = "<ACTION>" });

        return app;
    }
}
```

UmbracoApplicationBuilderExtensions

Within a Umbraco application, you reference the routing builder within `Start` inside of `.WithEndpoints()` method `

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseUmbraco()
        .WithEndpoints(u =>
        {
            u.UseCustomRoutingRules();
        });
}
```

Adding Rules To Start.cs

Separating all your custom routing rules like this will give you a nice clean separation of logic. Without some careful pre-planning, it can be very easy for `Start` to turn into a god classes. When you can contain all the routing rules within a related file, you are fine.

Route Hijacking

When working with controllers within Umbraco, a term that you will quickly bump into is route-hijacking. Before we deep-dive into all the different types of controllers, it is important that you understand what this term means.

Within a vanilla .NET website, we use controllers, models and views to render pages. When you are building a Umbraco website, it is possible to omit the controller. In fact, within Umbraco, the default mode is to not use a controller at all!

When you create a new document type, it is possible to enable an option for an associated template to be created as well. Within Umbraco terminology, a template means a .NET view. When a page match has been made, the default Umbraco routing rule will first check for a corresponding controller. If one is not found, the next rule will look for a corresponding view.

You can prove how this default routing works yourself. Within the backend, go to `Settings`. Within the `Document Types` node, create a new document type. When creating that type, use the option `Document Type with Template`. Next, give the document type an alias, say `test` and click `Save`. If the process has been completed successfully, within the application `View` folder a new view will have been created called `test.cshtml`.

If you then created a page within the content tree using this new `test` document type and tried to view that page from the website, the HTML

within `test.cshtml` will be returned to the browser. Notice, within this process, you did not need to write any C# code at all!

This default routing process has pros and cons. For a developer, the pros are questionable, however, for non-technical people, having the ability to build pages without knowing C# can be useful.

If the website owner only has access to people who know HTML and Javascript, using a view-only approach means that the team can update the website. They can do this without needing to know C# or even how to release the website. This is possible through Umbraco's in-built code editor.

You can access the code editor within the `Settings` section in the backend. Expand either the `Templates` or `Partial Views` node. Under either node, you will see a list of all the corresponding view files that have been created inside of the view folder. Select any view and Umbraco will load the contents of that view within its code editor. Using this editor, the HTML within the view can be tweaked and modified as needed.

The Umbraco code editor can be used to update any `.cshtml` files contained within the `View` folder directly. Some website owners will appreciate having this ability to update the HTML directly within the CMS, however, this approach definitely has some negative trade-offs:

- All changes to the HTML are being made directly on the server, meaning updates will not be in source-control
- The code editor has no version control support. Make a mistake and you can not roll back!
- Only simple HTML changes can be made in a view. If you need complex logic, your screwed!
- Changes in the code editor will impact production instantly. Exposing customers to on the fly development work will negatively impact their site experience
- If the site uses caching, testing can be hard!
- No unit tests
- No ability to do more complex things like caching, validation, etc…

You might wonder how this routing magic works behind the scenes? Out-of-the-box, Umbraco ships with a controller type called `RenderController`. We will cover `RenderController` in more detail next.

In terms of this automatic routing magic, the default Umbraco routing rule is configured to route all page requests to the `Index` action defined within `RenderController`. Based on the name of the document type, the code within this base `Index` action will parse the incoming request and attempt to load a corresponding view. The code inspects the incoming contents document type and uses its alias as the key for the associ-

ated view name. This is why by default, the name of the view file (`.cshtml`) needs to mirror the document type alias!

It is also possible to override this default behavior by implementing individual controllers on a per document type basis. From a good software architecture perspective, allowing Umbraco to directly load a view is suboptimal. The benefit of using a controller rather than calling a view directly should hopefully be obvious to the developers reading this:

- Version control
- C# logic is added at the controller level rather than a view
- Ability to use view models
- Code added within a controller can be unit tested
- Complex logic can be triggered inside the controller
- Caching, anti-forgery tokens, etc… can be applied to the controller

This is where route-hijacking saves the day! Route-hijacking involves overriding this default routing process and associating specific controllers on a per document type basis. To map a document type to a controller you need two things. First, you need to create a new controller in code which inherits from the correct base type. Secondly, you need to name it correctly. The Umbraco naming convention requires you to use the document type alias as the prefix to the class name, followed by the word `Controller`. The `Controller` part is a .NET convention rather than a Umbraco thing. To give you some examples:

- An alias of `Home` would map to a controller called `HomeController`
- An alias of `LandingPage` would map to a controller called `LandingPageController`
- An alias of `Search` would map to a controller called `SearchController`

Umbraco uses route-hijacking routing to automatically wire up the correct routing. As a developer, you do not need to add any routing rules within the routing table to work with Umbraco content. Follow the steps mentioned above. Make sure you do not have any typos, otherwise, say hello to a 404 error!

The main takeaway from this section, is that using controllers within your project is a good thing. For normal Umbraco content, routing will automatically work due to route hijacking.

Next, let us look at the different types of controllers that you have at your disposal and how you can go about creating each one. We will start with `RenderController`.

Render Controller

The controller type that you will be using most frequently within an Umbraco build is the `RenderController`. The reason why

`RenderController` is the most commonly used controller type within a Umbraco project is that the `RenderController` type's main purpose is to render Umbraco pages.

Whenever you want to associate a controller to a document type, you will need to create a controller that inherits from `RenderController`. The good news is that you will never need to worry about adding custom routing rules when using `RenderController`. Create the controller, job done!

The code to create a custom `RenderController` controller is very similar to the code required to create a vanilla .NET MVC:

```
public class HomeController : RenderController
{
    private IPublishedValueFallback _publishedValueFallback;
    private IBlogService _blogService;
    private IMenuService _menuService;

    public HomeController(
            ILogger<HomeController> logger,
            ICompositeViewEngine compositeViewEngine,
            IUmbracoContextAccessor umbracoContextAccessor,
            IPublishedValueFallback publishedValueFallback,
            IBlogService blogService,
            IMenuService menuService
        )
        : base(logger, compositeViewEngine, umbracoContextAccessor)
    {
        _publishedValueFallback = publishedValueFallback;
        _blogService = blogService;
        _menuService = menuService;
    }

    public override IActionResult Index()
    {
        var home = new Home(CurrentPage, _publishedValueFallback);
        var viewModel = new ComposedPageViewModel<Home, HomeViewModel>
        {
            Page = home,
            ViewModel = new HomeViewModel
            {
                Blogs = _blogService.Blogs
            }
        };

        return View("~/Views/Home/index.cshtml", viewModel);
    }
}
```

Render Controller

One big difference in `RenderController` in v9 compared to v8 is that there is no parameterless constructor. As you can see from the snippet, you need to inject a number of Umbraco related dependencies whenever

you use `RenderController`. You will then need to pass those dependencies into the `base` constructor of `RenderController`.

This is one of the more noticeable changes between Umbraco V9 compared to V8. This change means writing slightly more code when creating a new controller, however, not having the parameterless constructor makes the unit testing process more consistent.

The other notable difference in the snippet above compared to the code to build a vanilla MVC `Controller` is remembering to add the `override` operator when defining `Index`.

Asides from those two differences, everything that you can do within a normal controller, is possible within `RenderController`. You can apply the same attributes, create different action methods and customise things until the cows come home!

Let us recap before moving on. You create a document type within the CMS. That document type will be given an alias. To render a page of that type you create a new controller that inherits from `RenderController`. The name you assign that controller should be the document types alias followed by the term `Controller`.

Whenever a site visitor tries to view any page on the frontend website that has been created using that document type, the `Index` method within that custom controller will be called. Within the `Index` action, you can add any logic that you need.

Finally, within the controller, you can specify which view (`.cshtml`) file should be associated with the request. Typically, you will create the associated view file within the `View` folder, within a folder. The name of the folder should match the document type alias. The name of the view file should mirror the name of the action method that called it. Typically this will be `Index`:

```
View > Home > index.cshtml
```

As you can see getting up and running with Umbraco is dead-simple. Umbraco deals with all the routing for you. All you need to do is add in the corresponding controllers and voilà!

Umbraco API Controller

During a build, you will often need to expose data via an API. Maybe you want to create a website using a headless architecture, maybe you need to share CMS content omni-channel, or maybe you need to expose data from a backend service. Whatever the use case, building an API is significantly different compared to rendering a CMS page.

The good news is that dealing with different types of rendering requests within Umbraco is not a problem. Asides from `RenderController`, Umbraco ships with an additional five types of controller. Each controller type has a very specific use case.

When you need to build an API within Umbraco, you can use `UmbracoApiController`. The benefit of using `UmbracoApiController` is that everything will just magically work. You will not need to create any rules within the routing table.

Architecturally, you are definitely not forced into using `UmbracoApiController` when building an API. If you really have your mindset on using one of the vanilla .NET types to build an API, do not panic, you still can. You can create a new controller that uses `ApiController` with `ControllerBase`. You could also use the normal MVC `Controller` type and decorate the actions with the appropriate REST attributes (GET/POST/PUT/etc…).

The downside of using either of these default .NET 5 controller types, is that you will need to create and maintain the routing rules yourself. When you use a vanilla controller within Umbraco, that controller will never trigger unless you have defined the correct routing rule.

The only real caveat from using `UmbracoApiController` is that the API Url structure will need to follow a certain convention. To get the goodness of auto-routing, your API URL will need to be prefixed with `Umbraco/Api/`. Asides from this nuance, the steps required to create an API using `UmbracoApiController` are exactly as you would imagine. As you can see below, the code to create a custom API controller is simple:

```
public class MyApiController : UmbracoApiController
{
    public string GetSomeData()
    {
        return "This is API Data";
    }
}
```

Umbraco Api Controller

The URL for this controller would be:

`https://www.website.com/Umbraco/Api/MyApi/GetSomeData`

The first two segments (`Umbraco/Api`) are mandatory. The last two segments, map to the controller name (`MyApiController`) and an action method (`GetSomeData`).

If you do not want your API to have `Umbraco/Api/` in the Url, you will need to use the vanilla .NET controller and manually create a rule in the routing table. The end output Url is the only differentiator between the two.

I hope that you can now appreciate the importance of understanding how Umbraco routing works. If you do not know the nuances in the routing and when you need to add a custom rule, the experience can be very frustrating!

Surface Controller

The next type of controller is useful when you need to create a web form on any of your pages. In terms of routing, the big difference with a web form is that will need to post data back to the server. Dealing with data being passed back to your website is where the `SurfaceController` shines.

One potential mental gotcha you need to understand when using `SurfaceController` is that it is not a page-level controller. `SurfaceController` works on a component/child/partial level. `SurfaceController` is not a replacement for `RenderController` or any of the page-level related controllers. You can not associate a `SurfaceController` to a document type directly!

As a `SurfaceController` works on a child action level, you always need to call a surface controller from within a Umbraco page. Not fully appreciating this difference definitely caught me out when I started learning Umbraco! Never attempt to directly route a page-level request to a `SurfaceController`, instead create a form on a page and then call the surface controller on the form submission action.

Dealing with a form submission will require passing data back to the CMS. As the form is contained on a page trying to figure out which controller do you need to send the request back to is tricky.

As the routing within Umbraco does not directly map to a controller/action, trying to call a second action contained within a `RenderController` is not easy. This is why creating a separate controller just to deal with the form postback makes life much easier.

Using the `SurfaceController` means you do not need to think about how that routing works at all. Set the surface controller and the action within the form submissions HTML and the routing just works. The code to create a surface controller is shown below:

```
public class MyFormController : SurfaceController
{
    public MyFormController(
        IUmbracoContextAccessor umbracoContextAccessor,
        IUmbracoDatabaseFactory databaseFactory,
        ServiceContext services,
        AppCaches appCaches,
        IProfilingLogger profilingLogger,
        IPublishedUrlProvider publishedUrlProvider)
        : base(umbracoContextAccessor, databaseFactory, services, appCaches, profilingLogger, publishedUrlProvider)
    {
    }

    [HttpPost]
    [ValidateUmbracoFormRouteString]
    public IActionResult Postback()
    {
        return Content("Hello Postback");
    }
}
```

Surface Controller

Notice the line that defines `ValidateUmbracoFormRouteString`. This attribute means only form data submitted from a Umbraco page will be permitted. This is useful for security and I recommend that you always include it!

For long-term Umbraco users, `SurfaceController` works a little differently in V9/10, compared to V8. The main difference is around how the form is rendered. In V8 you used to create two action methods in the surface controller, one to render the HTML and one to deal with the postback. In .NET Core, there is a new way to render components called `ViewComponents`. `ViewComponents` will be covered in more detail later in this chapter.

In V9/10, you create a `ViewComponent` to add the form HTML. You would then call that `ViewComponent` from a page to render the form. You would then create an additional `SurfaceController`. The job of the `SurfaceController` is to just deal with the form postback.

In the new world, you create two things, a view component and surface controller. In the land of V8, you only created the surface controller. Let us look at the code to make this a little clearer. First, you would create a `ViewComponent` controller with an associated `cshtml` view to render the form. You could even create the form as a component that can be selectable using the `BlockListEditor` property type! In that view, you need to add the HTML that posts the form data back to the `SurfaceController`. This is done like this:

```
@using (Html.BeginUmbracoForm<JonDJones.Core.Controller.MyFormController>("HandleSubmit"))
{
    <input type="submit" />
    <button>Submit</button>
}
```

Surface Controller Form Example

To post a form to a `SurfaceController` you need to use `BeginUmbracoForm` where `T` is the surface controller type you want the form to post back to. `BeginUmbracoForm()` takes a single parameter which is the action within the controller you want the submission to be sent. In this example, the action is called `HandleSubmit`. That's all you need to do in order to start writing custom forms in Umbraco!

Authorized Controllers

Up until this point, the main purpose of each controller type has been to render some form of data to the frontend. Not all data should be accessible to everyone, it's time to talk lockdowns (not COIVD!).

Umbraco also provides several secure controller types. These controller types are useful for backend development. Umbraco provides controllers for both secure API and page-level access. The most important thing to note about creating any secure controller within Umbraco, is that you also need to adhere to Umbraco specific rules around the Url format. To make a Url secure within Umbraco, it must be prefixed with `/umbraco/backoffice`.

We will start by looking at `UmbracoAuthorizedApiController`.

Umbraco Authorized Api Controller

The `UmbracoAuthorizedApiController` type can be used to create secure API end-points that only logged-in CMS users can access. `UmbracoAuthorizedApiController` is perfect for when you need to build a custom backend screen where the data is sensitive.

The nice thing about `UmbracoAuthorizedApiController` is that all the routing is taken care of for you. Just like `UmbracoApiController`, you will not need to add any rules to the routing table. The code to create a secure API is simple enough and looks like this:

```
public class BackendController : UmbracoAuthorizedController
{
    public IActionResult Index()
    {
        return Content("Hello from authorized controller");
    }
}
```

Umbraco Authorized Api Controller

To access a secure controller, remember, you need to prefix the Url with `/umbraco/backoffice`. Additionally, with `UmbracoAuthorizedApiController` you also need to add an additional segment called `API` to the Url as well. Meaning, the complete URL prefix for `UmbracoAuthorizedApiController` is `/umbraco/backoffice/api`.

To trigger the API defined above, you would use this URL `/umbraco/backoffice/api/secure/index`. Where the `secure` segment maps to the controller name and the `index` segment maps to the name of the action within the controller. Asides from that this controller type works exactly the same as `UmbracoApiController`.

Umbraco Authorized Controller

The final Umbraco controller type, `UmbracoAuthorizedController`, allows for secure page-level CMS only access. Again, as UmbracoAuthorizedController`is a secure controller, the URL needs to be prefixed with`/umbraco/backoffice/`.

Unfortunately, `UmbracoAuthorizedController` does not auto map any routing rules for you. When using `UmbracoAuthorizedController`, you will need to define your own custom rule in the routeing table to allow a URL to trigger the controller. The code to create a controller is very similar to normal `RenderController`:

```
public class BackendController : UmbracoAuthorizedController
{
    public IActionResult Index()
    {
        return Content("Hello from authorized controller");
    }
}
```

Secure Routing Rule

As you can see from the code, there is nothing to be concerned about here. The important thing to remember when using `UmbracoAuthorizedController` is that the controller and action names need to map correctly in the corresponding routing rule. Adding a secure rule for this controller would look like this:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseUmbraco()
        .WithEndpoints(u =>
        {
            u.EndpointRouteBuilder.MapControllerRoute(
                        "secure-route",
                        "umbraco/backoffice/Plugins/Backend/Index",
                        new { Controller = "Backend", Action = "Index" });
        });
}
```

Secure Routing Rule

I had trouble getting `UmbracoAuthorizedController` to work with a view. When I tried to return a `View` from this controller I encountered this error:

```
The name 'View' does not exist within the current contex
```

If you encounter a similar issue it is also possible for you to manually create a secure controller yourself. If you look at the source code (here), you will see the `UmbracoAuthorizedController` is just a normal controller that decorates two attributes. We can clone its functionality like this:

```
[Authorize(Policy = AuthorizationPolicies.BackOfficeAccess)]
[DisableBrowserCache]
public class AdminController : Microsoft.AspNetCore.Mvc.Controller
{
}
```

Admin Controller

This controller will do exactly the same as `UmbracoAuthorizedController`.

Controller

Asides from the controllers supplied by Umbraco, it is still possible to use the normal vanilla MVC `Controller` type. The vanilla MVC controller is useful when you need to render pages on the frontend that do not rely on content defined inside of Umbraco. Some classic content modelling use cases where this type of need can occur include building an RSS feed, or creating an XML sitemap. The code to create a normal controller in Umbraco is the same as any vanilla MVC powered project:

```
public class VanillaController : Microsoft.AspNetCore.Mvc.Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Vanilla MVC Controller

The trickier part of creating a normal MVC controller within Umbraco is the routing. As we know, Umbraco overrides the normal MVC routing rules with its own custom rule. If we want to use a normal MVC controller in Umbraco, you will need to create your own custom routing rule before it will trigger. The rule for a normal `Controller` is the same as the previous examples:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{

    app.UseUmbraco()
        .WithEndpoints(u =>
        {
            u.EndpointRouteBuilder.MapControllerRoute(
                        "vanilla-route",
                        "/vanilla/{action}/{id?}",
                        new { Controller = "Vanilla", Action = "Index" });
        });
}
```

Routing Rule

With this rule enabled, when a request to `www.website.com/vanilla` is made, the routing rule will redirect the request to the `Index` action within the `VanillaController`.

View Components

When rendering pages within Umbraco, you will often need to render components contained within those pages. Some of these components will be Umbraco based components, e.g. blocks being rendered within the `BlockListEditor`. Other components could be more UI focused like a menu, a breadcrumb, the header or the footer.

Ideally, when rendering components you will still want to associate a controller to a component in order to have complete control over how that component is rendered.. Unfortunately, you will not be able to use any of the out-of-the-box Umbraco controllers to accomplish this. The Umbraco controller are mainly aimed at page level rendering.

Historically within ASP.NET MVC, the mechanism of getting the data required to render components was not as optimal as it could have been. In order to render a component you had two paths you could take. The most common pattern was to get all the data required to render all components referenced by a page within the page controller. All the processing was done within the controller's default action method. A giant view model would be generated that contained all page and component data. This view model would then be passed into the associated page view.

Components could be rendered from the page view using partial views. The main purpose of a partial component was to just render the model it received. Partial views would take a view model and render some HTML, simple.

Code wise, this architecture was OK but not perfect. Having to gather and compose the data required to render the same components within multiple page controllers could be annoying. To get around this frustration, developers often came up with various patterns to make the component view model creation process more efficient.

Some developers would create component level services that would be injected into the page controller. Others used inheritance within the view model to standardize the codebase. Architecturally, this was OK as it allowed teams to adhere to the MVC paradigm, however, with a slight trade-off.

Another pattern to get around this limitation, was to use the `HTML.Action()` HTML helper method combined with a `ChildAction`. In my last book, I recommend you adopted this pattern within the global layout. Here is a code snippet is taken directly from it:

```
<!DOCTYPE html>
<html lang="en">
    <head>
    </head>
    <body>
            @Html.Action("RenderHeader", "SiteFurniture")

            @RenderBody()
    </body>
</html>
```

View Component Example

This pattern typically resulted in better re-use, as it allowed the code related to the component to be moved from many page controllers into a single component controller. This pattern also had limitations.

First, the calling code is dependent on the correct string name being provided. As a string is being used as the identifier rather than a strongly-typed reference, there is no intellisense or compiler checking built-in.

After launching your site, if you ever needed to refactor the component, using a string value as the component identifier could result in accidental misreferences. When you can not rely upon the compiler to inform you about all of the locations a component is used, it is very easy to forget all the areas where a component is referenced!

The second issue with this pattern is that it was never designed to be as performant as it could have been. Any requests made to render a child action are not natively asynchronous. All child action calls were essentially page-level render-blocking calls. As the call to render the component from a page was being made, the page processor was sat twiddling its thumb until the component finished parsing!

The concept of the `ViewComponent` was introduced in ASP.NET Core MVC and onwards. You can think of `ViewComponent` as a replacement for `ChildActions` and an upgrade to `partial` views.

ViewComponents are completely self-contained objects that can be used to render HTML from a razor view in a consistent way. A `ViewComponent` can be associated with a controller. Meaning, that calling a `ViewComponent` offers the same benefits as calling a page controller. All your component code can be contained within a single-area promoting re-use. You can apply caching, permissions and anything else you see fit!

`ViewComponent` supports both `async` and parallelism. Combined with its useful overload and Tag Helper, its possible to strong-type it and get IntelliSense to work within the component declaration code!

The code to create a `ViewComponent` is straight-forward and an example can be seen below:

```
public class MyViewComponent : ViewComponent
{
    public MyViewComponent()
    {
    }

    public async Task<IViewComponentResult> InvokeAsync()
    {
        return await Task.FromResult();
    }
}
```

View Component Example

In order to call this view component from a Razor view, you would use this syntax:

```
@await Component.InvokeAsync("MyViewComponent")
```

Invoking A View Component

Using the code above you can call a component anywhere from within a page. One downside of this invocation code is that you still have to manually pass in the string identifier. Luckily, `InvokeAsync()` also comes with an overload that takes the a type as an argument. The code above can be refactored to make it a little nicer like this:

```
@using Namespace.ViewComponents;

@await Component.InvokeAsync(typeof(MyViewComponent))
```

Invoking A View Component

I recommend you favour `ViewComponents` as the way to render normal components as well as Umbraco components.

Updating BlockListEditor to work with view components

The code above is great for building normal components within an Umbraco propjet, however, by default the `BlockListEditor` property has not been designed to work with `ViewComponents`. As you wil likely be using `BlockListEditor` to render most of your components, you will want to change the properties default rendering behavior to get all these wonderful benefits! Luckily, there is a very easy to fix to solve this.

The `BlockListEditor` property makes uses of a file called `default.cshtml` in order to render any components added by content editors within the property. This view file can be found within your solutions `Views` folder in this location:

`Views > Partials > blocklist > default.cshtml`

If you open this file within your IDE, within the `foreach` loop you will see a line that tries to call a partial view for each component directly:

```
@await Html.PartialAsync("blocklist/Components/" + data.ContentType.Alias, block)
```

Default BlockListEditor View

You can change this line to call a view component controller instead. The code to do that is shown below:

```
@await Component.InvokeAsync(data.ContentType.Alias, (IPublishedElement)data)
```

Calling A View Component Controller Within BlockListEditor

To make the block-list editor render a view component, you need to change the HTML to use the `InvokeAsync()` method. To render the correct view component for each added block, you will need to pass the current blocks alias as the first argument into `InvokeAsync()`.

Applying this change, means you will need to create a corresponding view component for each block you create within the CMS. The naming convention of the view component will based on this alias. For example, a

block with alias of `banner` would map to a view component called `bannerViewComponent`. Remember, blocks are defined as document-types marked as `Elements`!

For the second parameter, you will need to pass the current block object back to the controller as a type `IPublishedElement`. For completeness the final updated code for `default.cshtml` would look like this:

```
@inherits Umbraco.Cms.Web.Common.Views.UmbracoViewPage<Umbraco.Cms.Core.Models.Blocks.BlockListModel>
@{
    if (!Model.Any()) { return; }
}

@foreach (var block in Model)
{
    if (block?.ContentUdi == null) { continue; }
    var data = block.Content;

    @await Component.InvokeAsync(data.ContentType.Alias, (IPublishedElement)data)
}
```

Calling A View Component Controller Within BlockListEditor

After making these two changes, when a request to render a block-list editor property is made, the property will try to call the corresponding view component for each block. Assuming all the naming conventions match, the request will be automatically wired up. The current block object will then be accessible from within the view component:

```
public class bannerViewComponent : ViewComponent
{
    private IPublishedValueFallback _publishedValueFallback;

    public bannerViewComponent(
            IPublishedValueFallback publishedValueFallback)
    {
        _publishedValueFallback = publishedValueFallback;
    }

    public async Task<IViewComponentResult> InvokeAsync(IPublishedElement model)
    {
        var banner = new Banner(model, _publishedValueFallback);
        return View(banner);
    }
}
```

Banner View Component Example

As you can see inside of the `InvokeAsync()`, the `IPublishedElement` is passed in and then from within the code you then have access to the CMS data in code. Do not worry at the minute if this code is confusing, it will be covered in detail shortly. This change while trivial will allow you to

create a much more structured codebase that applies to the MVC paradigm.

Umbraco Models Builder

After successfully setting up route-hijacking and connecting a controller with the corresponding type of content, the next logical step is to write the code to access the related CMS data. Regardless of whether the request is a page-level request or a block-level request, the optimal way to work with CMS data in code is to use auto-generated C# objects based on the related document type.

The focus of this section is on writing the code required to access the content related to the current request. It is also very likely that you will also need to access content contain in other CMS pages. This type of access is possible via the Umbraco APIs. In the next chapter, you will learn more about these APIs and how to query the CMS in order to get any data you require. Mastering these two topics should allow you to build the majority of the elements that you identified during the content modelling process.

As we have just seen in the last section, it is possible to use objects created from C# classes to access CMS data in code. This is made possible using the Umbraco Models Builder.

The history of Umbraco Models Builder is interesting. Originally developed as a community package by Dave Woestenborghs. Models builder was then ingested into the core Umbraco product around version 8.

My personal preference for using the Model Builder is to use it to generate C# classes inside of a dedicated class library, however, this is not the only way that Model Builder can be configured. Before getting to the code, we will start by taking a deeper dive into the capabilities Model Builder provides.

What Is The Umbraco Model Building?

The sole purpose of Model Builder is to convert the document types created inside of the CMS into models that can be used in code. Note that I specifically wrote models and not C# classes. Models Builder has three different conversion modes. One mode generates virtual models, the others C# classes:

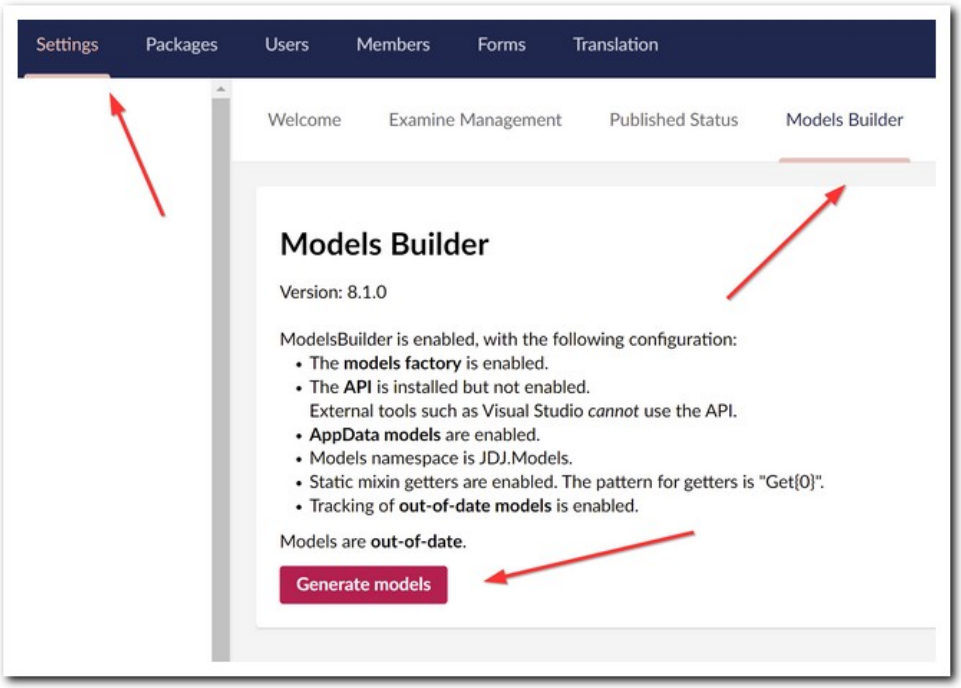
InMemoryAuto: `InMemoryAuto` is the default mode of operation for the Models Builder. As the name implies, `InMemoryAuto` mode generates the models virtually within memory. Whenever a content editor makes a change to a document type within the CMS, models are generated and compiled on the fly

The benefit of using `InMemoryAuto` mode is that the models are updated without having to either recompile or rebuild the website. The downside of `InMemoryAuto` mode is that you can not access these models within Visual Studio.

At this point in the book, if you hate the idea of route-hijacking and you have your heart set on building the entirety of your website from inside of the CMS, use `InMemoryAuto` mode. If you want to use Visual Studio to create and manage the website, use one of the other modes.

SourceCodeAuto and SourceCodeManual: The other two modes, `SourceCodeAuto` and `SourceCodeManual` work very similarly. In both modes, C# classes are generated that can be used within Visual Studio. In both modes, after the models are generated, a developer will need to compile the solution before any updates become available for use in code. By default both modes will generate the models within `~/umbraco/models`. The difference between the two modes is when the models are generated.

- SourceCodeAuto: In `SourceCodeAuto` mode, C# classes are automatically generated whenever a content editor makes a change within the CMS.
- SourceCodeManual: In `SourceCodeManual` mode, the C# classes are only generated when a content editor clicks on the generate models button within the CMS. To access this button within the backend, navigate to the `Settings` section. You should see a tab labelled `ModelsBuilder`. Within this screen you should see the generate models button:



Generate models using Model Builder

As detailed within the installation chapter, my personal preference is to use Model Builder using `SourceCodeManual` mode. From my experience, the most efficient way to manage a Umbraco project is to agree for the development team to make all document type changes. Anytime a document type is need to be either created or updated, the change is always made on a local machine and then thoroughly tested before being released into production.

Updating/creating a document type is an action that is always part of a larger change. That larger change could involve updating controllers, view models, views, and hopefully unit tests. Using `SourceCodeManual` mode can help give teams more security as it means only developers can update the CMS.

Using `SourceCodeManual` mode also means that whenever you need to compile the project you do not suddenly encounter unexpected `surprises`. For example, if you and the team use a shared database for local development, it can be very annoying when someone updates the CMS and your code suddenly breaks for no reason! Surprises and best left for your birthday!

Using Model Builder in `SourceCodeManual` mode is my personal preference, however, like all these subjective configurations, pick the approach that makes you happy.

Setting the mode that you want Model Builder to run under is made within `appsettings.json`. After installing Umbraco you should see this section with `appsettings.json`:

```
Umbraco > CMS > ModelsBuilder
```

The mode type is set using the `ModelsMode` property. Asides from the `ModelsMode`, Models Builder can be further configured and tweaked using these properties:

ModelsNamespace: This property defines the namespace that will be used when the models are generated. `ModelsNamespace` is a non-mandatory property. Leaving it blank will mean the namespace will default to `Umbraco.Cms.Web.Common.PublishedModels`.

FlagOutOfDateModels: The `FlagOutOfDateModels` property is used to configure if the CMS should flag when the models become out-of-date and consequently need refreshing. This property is also not mandatory and will default to `true` if not explicitly set. I can not really think of a great use case to disable this feature, so I recommend ignoring it!

ModelsDirectory: This property defines the directory the models will be generated inside of. When setting this property, the important thing to remember is that the path needs to map to a virtual folder rather than a physical folder. In practical terms, this means that you will need to prefix any value you define with `~/`

AcceptUnsafeModelsDirectory: When the Model Builder has been set to use one of the source code modes, by default, it will generate all the models inside the current website directory, inside this folder:

`~/umbraco/models`

Using the `ModelsDirectory` property, you can change this location. If you tried to change the location to a folder that lives outside the webroot, you will encounter an error. This is where `AcceptUnsafeModelsDirectory` mode comes into play. When enabled, the builder will allow you to generate the classes anywhere within the solution folder.

Enabling this mode will allow the classes to be generated in the solution root, additional class libraries, or anywhere that takes your fancy. As enabling this mode is considered a risk, it is disabled by default.

This covers the main ways of configuring Model Builder, the next step is to use these models in code to instantiate objects from them!

Accessing Umbraco Content Via Models In Controllers

Assuming you opted to configure Model Builder to generate classes, you will also need to understand how to instantiate objects from these models. Each model generated from the model builder contains two mandatory constructor parameters.

The first one is of type `IPublishedContent`. `IPublishedContent` will be covered in a lot more detail within the next chapter, however, for now, think of it as a generic representation of any page, component, or media item that has been published inside of the CMS.

Our aim is to convert this generic object type and convert it into a very specific object type. For each document type created inside of the CMS, an associate model will be generated. Your job is to create a related route hijacked controller and then instantiate the related model inside that controller.

To make this work, you will first need to access the `IPublishedContent` representation of the current page within the controller. Whenever you create a controller than inherits from `RenderController`, you will get access to a property that this base type exposes called `CurrentPage`.

As you might guess, `CurrentPage` is of type `IPublishedContent`. `CurrentPage` will be automatically populated on a successful route-hijack with all the related page data. Whenever you want to instantiate a model based on the current page from Model Builder, you can simply pass `CurrentPage` as the first constructor parameter.

The second constructor argument is of type `IPublishedValueFallback`. Whenever you call a property contained on a document, under the hood an additional call to a class called PublishedValueFallback is made.

It will be easier to explain the usefulness of this second parameter with some code. The code below will get the value from a property called `myProperty` out of a document type created within an object of type `IPublishedContent`:

```
var prop = CurrentPage.GetProperty("myProperty");
var value = prop.GetValue("culture", "segment");
```

Getting A Property

The parameters being passed into `GetValue()` are both optional. These parameters can be supplied to define exactly how Umbraco should return data. Being able to set the `culture` and the `segment` is useful when building a project that makes use of Umbraco's multi-language capabilities. Setting the `culture` will set the language the content will be returned in.

The obvious issue with this code is duplication. Having to define these additional configurations individually throughout your codebase would lead to a lot of code smells. To avoid this nastiness, `IPublishedValueFallback` becomes handy.

The aim of `IPublishedFallback` is to allow you to define the default configurations Umbraco should use when retrieving values out of properties. Being able to provide default fallback options means you do not need to apply the same configurations over and over again.

Let us imagine you use a multi-language website. The default language is English, however, the site contains some translations in Spanish. When a site visitor browses the Spanish site, if no Spanish content instead of displaying a blank page, you might want the content to default to the English copy.

If the language-specific version of some content is missing, you can set a fallback language that Umbraco should use instead. Instead of having to define this fallback configuration everywhere in the codebase, you can define the configuration once within `PublishedValueFallback`.

Using `PublishedValueFallback` means less code to write overall. Be warned, there is a limited document around `IPublishedValueFallback`. If you try googling the term, you will not find much. The `PublishedValueFallback` class contains a number of properties including `content`, `alias`, `culture`, `segment`, `fallback`, `defaultValue`, and `value`.

If you are still struggling to see the usefulness of this type, I think you will gain the most understanding by looking at the code.

To be honest, unless you have a complex multi-language website, I doubt that you will ever need to override the default options. If you do ever encounter a need to change the default configuration, it will be for a partic-

ularly niche use case. What this means for day-to-day development is a lot of copy and pasting within the controller code when initiating models.

Combining `IPublishedContent` and `IPublishedValueFallback` means you can instantiate a new model that has been generated from Models Builder. The code to do this within a controller will look like this:

```
public class HomeController : RenderController
{
    private IPublishedValueFallback _publishedValueFallback;

    public HomeController(
            ILogger<HomeController> logger,
            ICompositeViewEngine compositeViewEngine,
            IUmbracoContextAccessor umbracoContextAccessor,
            IPublishedValueFallback publishedValueFallback,
            )
        : base(logger, compositeViewEngine, umbracoContextAccessor)
    {
        _publishedValueFallback = publishedValueFallback;
    }

    public override IActionResult Index()
    {
        var home = new Home(CurrentPage, _publishedValueFallback);
        return View();
    }
}
```

Models Builder Generation

Getting the default `IPublishedValueFallback` object is done using dependency injection. Pass the type in as a constructor argument and off you go. `CurrentItem` is used to get the first property. After instantiating a new model, you can use strongly typed goodness to access CMS data. The code to access a property goes from that chunky ugly example above, to this:

```
var home = new Home(CurrentPage, _publishedValueFallback);
home.MyProperty
```

Models Builder Code

View Model Patterns

The aim of this chapter is to provide you with enough knowledge to render a basic web page within Umbraco. To accomplish that goal, we need to convert the data added by content editors within the CMS into HTML which is finally returned to a client's browser.

Combining route-hijacking within a custom controller will allow you to intercept any request to render a page or a block. You can access any related CMS content in code using the models generated by the Models Builder. The final hurdle is how we render the HTML?

To render HTML following the MVC paradigm, we have controllers, models and views. In this chapter we have the controller part nailed. In this section, we will focus on models and views.

The model part of MVC is concerned with creating an object that contains all the data required by the view layer (HTML payer). To create a nice separation of concerns, you should avoid writing any code within a view directly.

The issues with writing lots of code directly within the HTML layer are well documented, so I will not rehash them here. If you write all the code within your views, you may as well not use route-hijacking in the first place. To follow good software engineering principles, you need to create a view model within a controller that contains all the data required by the view.

You may have easily assumed that the models generated by the Umbraco Models Builder would also be used as the view model within Umbraco MVC, however, I do not recommend that. Using the models generated by the model's builder as the bridge between the controller and the view would be a mistake.

Within every controller that you create, you will likely want to write some code that either manipulates or cleanses the incoming CMS content before passing it down the pipeline. For example, if you need to render a date on a page, you will want to convert the not so human-friendly output of the `DateTime` property into something more human-readable. It is considered a better practice to write the code to do this conversion within the controller layer. In other situations, you may need to query the CMS for additional data, or, even call a third-party service or API. It is an incredibly rare situation when you do not need to perform any additional logic within a controller.

In .NET MVC, you can only pass one object from a controller into a view. If you use the Models Builder object as the view model, you would not be able to pass any additional data you create within the controller down into the HTML layer. If you used the Models Builder model as the view model, your only option to access additional data within the view would be to write the code directly within the HTML layer, yuk!

The point that I am trying to make is that within your controller, you will have one object that contains all the content added by the content editors. You also likely need additional data within the view layer. To get this data, you will write code within the controller. If you used the object generated by the Models Builder as your view model, you will have no way of passing this extra data into the view. This means we need to pick a strategy on how to resolve this conundrum!

Passing a view model down from a controller into a view might sound simple. however, there are a surprising number of patterns that you can pick from. Do you use inheritance or composition? Do you extend the Models Builder classes? Do you clone existing data into a new object? Do you do something completely different?

In this section, I will cover the most used data passing patterns so you can pick the approach that you find most optimal. To get started, we will first look at the code to pass the default Umbraco page object into the view as a starting reference. We can then improve upon that.

IPublishedContent Pattern

The quickest way to render CMS content within a view is to pass the `CurrentPage` property, which is of type `IPublishedContent`, down into the controller's corresponding view. The controller code to pass the `CurrentItem` into a view would look like this:

```
public override IActionResult Index()
{
    return View(CurrentPage);
}
```

Passing IPublishedContent into a view

The corresponding HTML within that view could then look something like this:

```
@inherits Umbraco.Cms.Web.Common.Views.UmbracoViewPage<Umbraco.Core.Models.PublishedContent.IPublishedContent>

@Model.Value("myProperty")
```

Rendering IPublishedContent within a view

Even though this code is simple, its not optimal. Lets consider some of the issues you would encounter using code like this within our projects. First and most importantly, passing an object of `IPublishedContent` directly into a view means that there is no way of passing additional data from the controller into the view.

If you needed to render some data or some content within the view that is not directly exposed by the `CurrentItem` object, you would need to write C# code within the view in order to get it.

The second issue with this approach is around data access. `IPublishedContent` is used to represent any document type, member, or an asset added within the CMS. `IPublishedContent` is a generic type that exposes generic methods that will allow you as a developer to access all the properties you need.

Accessing CMS data from an object of type `IPublishedContent` is achieved by passing in the property name you want to access. This is done by passing in the property name as a `string` into the `GetValue()` method.

Hardcoding property identifiers within your codebase will cause you maintenance headaches. If a content editor decided to change the property alias for any existing document type, the second the save button was pressed within Umbraco, any code that referenced that property would automatically break.

Your headache will be a result from not having an easy way to know where that property is referenced in the codebase easily. Hardcoding property references within your views will mean that you will not get any help from the compiler whenever a property alias is refactored. It will be left to good ole' search to help you manually find all the references. If you accidentally make a typo in the reference , the search function will not help. This is your problem pal!

Finally, when using objects of `IPublishedContent` there is no easy way when writing the HTML to easily find out what properties are available from the current document type. Expect a lot of switching to and fro from Visual Studio to the CMS to find out!

Models Builder Pattern

An alternative approach to level up the `IPublishedContent` pattern, is to pass the related model generated by Models Builder into the view. Us-

ing a strongly-typed object within your view will give you compiler safety and intellisense . This small change will make life much easier compared to working with a generic `IPublishedContent` type. The code to instantiate and return a models builder model is shown below:

```
@inherits Umbraco.Cms.Web.Common.Views.UmbracoViewPage<Umbraco.Core.Models.PublishedContent.IPublishedContent>

@Model.Value("myProperty")
```

Passing Models Builder object into a view

The corresponding view code would look like something this:

```
@inherits Umbraco.Cms.Web.Common.Views.UmbracoViewPage<Umbraco.Core.Models.PublishedContent.IPublishedContent>

@Model.Value("myProperty")
```

Rendering Models Builder object within a view

This pattern while an improvement, does not solve the issue of passing additional data into the view. The only data you can access in the view must be defined within the corresponding document type. To access any other data in the views you need to write C# code within the HTML layer. Not ideal.

The way I get around this limitation is to use a pattern that I made up that I named the Composed View Model Pattern.

Composed View Model Pattern

A limitation within .NET MVC is that you can only pass one object into a view. By creating an intermediary object, it is possible to pass two objects into the view. In the composed view model pattern, we create a base object that uses generics. The base object exposes two properties. The first property is used to return the object created by the models builder. The second property is used to return an additional view model. You can use this additional view model to pass any other non-CMS exposed data into the view. The code to create this base object is shown below:

```
public class ComposedViewModel<TPage, TViewModel> where TPage : PublishedContentModel
{
    public TPage Model { get; set;  }

    public TViewModel AdditionalData { get; set; }
}
```

Composed view model definition

The important part of this class is that you have a base object that uses generics to expose two properties of type `T`. You can name the class and properties whatever you like. To use this object to bind to the model's builder object and a view model, you will write code similar to this:

```
console.log('0. Run command `Polacode ')
console.log('1. Copy some code')
console.log('2. Paste into Polacode view')
console.log('3. Click the button ')
```

Using composed view model in code

In this example, `Blog` is the model generated by the model's builder and `BlogViewModel` is an additional view model that I created. `BlogViewModel` is nothing special, its just a class with some getters and setters (known as a POCO). Any additional properties that you need to use within the view should be added within this additional view model as a public property.

This pattern has lots of positives going for it:

- You get strongly-typed goodness and avoid hardcoding property names.
- Its easy to update and maintain. Need to drastically change how the page behaves later, simply create a new view model!
- Using this pattern will also mean that you will need to write less boilerplate code while unit testing. The reason why unit testing code will be easier is that you only really need to test the custom aspects of your view model and controller. There is little point in testing the models generated by Models Builder. By abstracting the custom code to a separate object, you only really need to test the code that populates that object!

To access properties exposed by the view model within the HTML using this pattern is very declarative. It is very easy to see where the data is coming from:

```
@model ComposedPageViewModel<Blog, BlogViewModel>

@Model.Model.CmsProperty
@Model.ViewModel.MyCustomProperty
```

Rendering composed view model in HTML

I think this code is very self-explanatory. If the rendered property is prefixed with `Model` then you are working with the Models Builder object and its CMS content. If the code is prefixed with `ViewModel` it is custom data that is set within the controller code.

This pattern may seem straightforward and insignificant, however. on a large project, this simple approach can significantly reduce code maintenance up-keep time by making the codebase simpler. This pattern is not the only pattern that you can use, however, this is the one I personally recommend.

Dedicated View Model Pattern

In the dedicated view model pattern, you create a new view model specifically for each controller. Within the controller's default action method, you have to manually map all the properties within the Models builder object that you want to expose within the view into this new view model. Any additional content that needs to be rendered on the page that is not contained within the Model Builder object can also easily be added to the new view model.

This pattern definitely gives the codebase a good separation of concerns. Code stays in the controller layer, HTML in the view later. You also get to use strongly-typed models so you get compiler safety and intellisense. Adopting this pattern is definitely better than simply passing a Umbraco object directly into a view, however, it has one big annoyance.

The big obvious downside of this pattern is upkeep and maintenance. Constantly having to map properties from the Models builder object into the new view model is a time suck. Whenever a document type is updated within the CMS, you will likely need to update your code. Even if the property is no longer used in the view, if it is being mapped to the view model, you will still need to fix it!

To reduce this overhead, the developers who favour this pattern tend to install AutoMapper. AutoMapper can be used for the heavy lifting of data copying from the Models Builder object into the dedicated view model object automatically.

AutoMapper can work well, however for me, it is just an extra package that needs to be installed and extra mapping code to maintain. Using the composed view model pattern means you do not need to install anything extra or write any mapping code. ### Partial View Model

Another possibility to allow for extra data to be passed into the view layer is to extend the classes generated by Models Builder. All classes generated by the Model Builder are created as a `partial` class.

If you have not come across the `partial` modifier in C# before, the premise is simple. With this modifier set on a class, it is possible to extend it. Create a new class with the same name and namespace as the class you want to extend. Set the new class with the `partial` modifier and the .NET runtime will treat it as a single class!

In this Partial View Model pattern, you only pass the model's builder object into the view, however, using an additional partial class, you extend the autogenerated objects behaviour to include any additional data that you might want to pass into the view instead.

Inheritance View Model Pattern

The final pattern that I will cover is probably the most used pattern out in production. Remember, just because a pattern is well used does not make it the best!

In this pattern, a new base view model is created that uses generics. There are a few variations to this pattern but the basic gist is to create a base view model like this:

```
public class BaseViewModel<T> where T : PublishedModels
{
    public BaseVM(T currentPage)
    {
        Page = currentPage;
    }

    public T Page { get; set; }
}
```

Base view model definition

The base view model uses generics to help reduce boilerplate set-up code. Instead of having to write the same set-up view model code over and over, you move it to the base view model instead.

The code to define a new view model based on a document type would look like this:

```
public class BlogViewModel : BaseViewModel<Blog>
{
    public BlogViewModel(ViewModels current)
        : base(current)
    { }

    public bool MyProperty { get; set; }
}
```

Defining a blog view model

The view model can then be instantiated with this code:

```
public override IActionResult Index()
{
    var viewModel = new BlogViewModel<Blog>();
    return View(viewModel);
}
```

Instantiating a blog view mode

This pattern is used extensively in the Umbraco world, it definitely works. The reason I do not like this pattern is that it uses inheritance. As first quoted in the Gang Of Four book, favour composition over inheritance! The reason for this mantra is that inheritance is harder to change. If you can compose properties onto a view model you will have more flexibility and options later on.

All the patterns listed above will work. The important part is the ability to use strongly-typed models and avoid writing code in the HTML. As long as you have those two boxes ticked, the pattern you use will be determined by your personal preference.

There are also other alternatives to these patterns that I have not covered. I personally recommend the composed pattern. For your project, pick the pattern that makes you happy!

Further Reading

- Ultimate Umbraco V9 Controller Guide
- A practical example of route hijacking in Umbraco
- Routing in ASP.NET Core
- Tag Helper information
- IPublishedValueFallback information
- What is route-hijacking
- Umbraco Models Builder guide.]

Umbraco API 101

In this chapter, we will deep-dive into the Umbraco APIs. During a build, you will often need to query the CMS in order to get additional content and data. The most obvious use case is getting the data required to render the header and footer. The header and footer content will obviously not be duplicated within the CMS on every page, so how do you access it in code?

Whenever you encounter a scenario where you need to access content that is external to the current request, you will need to perform an additional query to Umbraco to get that data. This raises the question, how do you query the CMS in the first place! Do you query the CMS using an ID, a Url, or, the page name?

The good news is that as long as you have a solid understanding of the content publishing process within Umbraco, querying the CMS for content is pretty straightforward.

We will start this chapter by discussing this process. As a quick high-level introduction, there are two types of content in Umbraco, published content and CMS content. There are fundamental differences between the two types. The underlining data types are different. The exposed properties and operations that are available by each type is different. Accessing the different types requires different APIs. To sum it up in three words, there are differences!

HTTP Context in ASP.NET Core

Before moving into the nitty-gritty detail of all the different ways you can query for content using Umbraco, it will be useful to talk about the `HttpContext` within .NET Core and how it has changed between .NET Framework.

The reason for discussing this framework change in relation to Umbraco is because some of the Umbraco APIs rely upon the `HttpContext` object.

Within Umbraco, we have `UmbracoContext` as well as `HttpContext`. `UmbracoContext` inherits from `HttpContext`. This means changes to `HttpContext` affect `UmbracoContext`. As `HttpContext` has changed in .NET Core, this is why accessing `UmbracoContext` has changed in v9 compared to v8.

The purpose of the `HttpContext` class is to 'encapsulate all HTTP-specific information about an individual HTTP request'. A new `HttpContext` object will be created whenever a new HTTP request or response is made to the application. The `HttpContext` object contains lots of data about the current request and it is an essential part of querying for content within Umbraco.

Each time a request is made to your application, a new `HttpContext` object is made. This object is then destroyed at the end of the current HTTP request or response. The `HttpContext` object contains information about things like the request headers, the response data, server information, session information, information about the current page object, cache information, as well as data about the requester like authentication and authorization status.

Within .NET Framework you could access the current `HttpContext` object using the `Current` property like this:

```
HttpContext.Current.Items.Add("Example", "SomeData");
```

Accessing HttpContext Within .NET Framework

Things have changed very slightly within .NET Core for `HttpContext`. The `Current` property is no more! Within any controller you can access the context object directly using this code:

```
public class HomepageController : Controller
{
    public override IActionResult Index()
    {
        HttpContext.Items["test"] = "test";
    }
}
```

Accessing HttpContext Within A Controller

If you need to access the `HttpContext` object outside of a controller, for example, within a custom service, you can access it via the `IHttpContextAccessor` interface.

Injecting `IHttpContextAccessor` as a constructor argument will give you access to a populated object of type `HttpContextAccessor`. The code to do that looks like this:

```
public class ExampleClass
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public ExampleClass(IHttpContextAccessor httpContextAccessor) =>
        _httpContextAccessor = httpContextAccessor;

    public void MyMethod()
    {
        var username = _httpContextAccessor.HttpContext.User.Identity.Name;
    }
}
```

Accessing HttpContext Within A Class

Asides from how to actually access the `HttpContext` object, everything else is pretty similar. The main takeaway is that the `Current` property has gone in .NET Core. This update to the core .NET framework is the reason why some of the data access methods within Umbraco v9 and upwards has changed slightly compared to the versions that used the .NET Framework.

IPublishedContent Vs IContent

To help you master Umbraco, it is essential that you have a good appreciation of how you may need to access content in different situations. Generally speaking, you will have two specific needs for accessing content within the CMS. One use-case will be to access content that has been published by a content editor for public-facing consumption. This consumption might be via a page, an API, or an RSS feed.

The other use-case will be for backend development. Out-of-the-box, a content editor can do lots of handy dandy things inside of the CMS. Sometimes you will bump into a requirement, or a request, for a content editor to be able to do something that is not possible using the features provided within the backend. If you ever find yourself in this situation, it is possible to extend Umbraco by building a custom backend section, screen, or view.As you will learn later on within this book, it is fairly easy to build your own custom backend screens within Umbraco.

When you encounter a requirement whose functionality will require you to extend the Umbraco backend, your content accessing requirements will also typically be slightly more complex. You may want to view the contents amend history, understand its publishing lifecycle, or modify the content in code and then persist that change back into the CMS.

An important distinction between these two scenarios is the content state. All of the frontend content will live within the Umbraco backend, however, not all of the content within the CMS will be published and live.

A content editor could in theory take several weeks to write a blog post. When the content editor decides to put virtual pen to paper, the blog post would get created inside the CMS, however, that post might not be public-facing for many weeks. Thinking in terms of state, the content will stay either saved or scheduled until it is published.

Within Umbraco, there is a fundamental difference between saved and scheduled content and published content. Understanding the differences between published and non-published states will save you a lot of frustration when working with Umbraco.

There are two big differences between yet-to-be published and published content within Umbraco. Whenever a page is published within the CMS, two important things happen that help improve performance. During the publishing process, a reference to the item will be added to the Umbraco search index. Umbraco ships with a search provider which is called Examine. Examine uses the Lucene search engine to allow for content searches to be made on CMS content. Examine and search will be covered in a later chapter, however, for a quick summary, if you want to write a site-wide search, querying data from this datastore will give you power and speed.

The second thing that happens during the items publish event is that a reference to the item is added to the Umbraco frontend cache. Within Umbraco terminology, this frontend cache is called NuCache. NuCache is used as the data source for many of the Umbraco service APIs to provide quick and fast access to published content.

Going back to our blog post example, if you wanted to create a custom backend screen that referenced the yet to be published blog post, you could not use the Umbraco service APIs that are populated by the data within NuCache. Querying this cache for non-published content would result in a no-match result.

Using a cache for accessing content means that Umbraco does not have to query the database. Any API that has to communicate directly to a database will not be ideal for performance. If you had to query the Umbraco database directly in order to process all page requests, your site would run extremely slowly. Without some form of cache layer, the server and the database would be very limited in the amount of traffic that it could handle. Not running an optimized website is financially expensive. Paying for additional servers just to deal with load is not a cheap architecture approach.

When you need to access public-facing content, you should always use the APIs that use the cache. You can tell when you are using a public-facing API because all the data returned will be of type `IPublishedContent`. In the last chapter, we encountered the `CurrentItem` property which was of type `IPublishedContent`. Based on this new perspective, you now know the query to populate this item was made to the cache, not the database.

Any request to get data from NuCache will be fast, however, there are some trade-offs. All public-facing APIs are read-only, meaning you can not modify their contents in code. Second, the cache only contains published

content. You can not access CMS content that has been saved or scheduled using the public-facing APIs. Lastly, you can only access the latest version of any content from the cache. Umbraco keeps a version history of all content changes made within the backend every time an item is saved. You can not access any version history using the public-facing APIs.

If you need to modify any content or, you need to access content that has yet to be published, you need to access code using a different API. It might help to think of this API as a backend API. This service API is called `IContentService`. You will know when you are using an API that queries the Umbraco database because it will return content as type `IContent`.

If you compare the public properties of `IPublishedContent` and `IContent`, you will notice some obvious differences. `IPublishedContent` defines some properties related to publishing. These include useful properties, like `PublishedDate`, `Url`, and `Segment`.

Conceptually when you think about it, this makes sense. If the content has not been published, why would it have a published date? In terms of writing code though, if you are trying to extend the backend and you are unaware of this limitation, you will waste many hours of Googling in vain wondering what you are doing wrong!

On the flip side, using the `IContentService` will give you access to operations on content that are not possible using the published content APIs. `IContentService` exposes methods like `GetVersion()` and `Save()` . When you are trying to extend the Umbraco backend, you will probably not get very far if you tried to access content via the published content APIs!

When you explain the differences between the two types of content within Umbraco, it logically makes sense, however, if you are unaware of these differences it can be a source of frustration. Using the wrong API and then getting confused as to why the data you want to access is missing is one of the most common mistakes people new to Umbraco make.

I have yet to encounter another CMS that has a similar architecture to Umbraco. I have had experience with around 15-20 different CMS systems since I became a professional coder. Most CMS architectures tend to only expose one content type and then provide a read-only and a read-write API.

This difference definitely threw me during my first Umbraco project. I accidentally used `IContentService` to deal with all public-facing content requests. In development when I was working on the site in isolation, I did not notice any performance impacts with `IContentService`. When the site was launched, the performance impact was noticeable extremely quickly! If you are new to Umbraco and you fall into this trap, do not sweat it, lots of other developers make the same mistake as well!

Now you conceptually understand the differences between the different content states, let us delve a little deeper into the published content APIs!

Published Content

When it comes to mastering the Umbraco published content APIs, I think that being mindful about how Umbraco has evolved over the years can be helpful. Umbraco was originally built using classic ASP around 2000. In 2004, Umbraco is updated to be fully .NET 1 compatible. For web development within ASP.NET 1, the web forms architecture was the only framework that was supported. For the old timers who remember, the web forms architecture was pretty cumbersome to work with. The web forms architecture was comprised of HTML, code-behinds and view state to build web pages from. The main takeaway from this history lesson is that the APIs were originally built to work optimally with web forms.

At some point in time (around v6), Umbraco was updated to support .NET MVC. The MVC architecture works nicely with dependency injection. Writing APIs to interfaces and then being able to access an API using dependency injection results in better architecture. This is why writing APIs to interfaces is considered more desirable.

When Umbraco MVC was first released, Umbraco still had to support dual architectures (web forms and MVC). This meant that none of the core services could be updated easily to support this more desirable way of working. As fewer people started using web forms, things started to change in Umbraco v8.

As we know in v9, the CMS was rewritten to fully support .NET Core. It has only been from v9 that every API has been updated to fully embraced `interface` first API access.

The takeaway from this little history lesson is that the CMS and the APIs have evolved over time. To support backwards compatibility, you can not easily rewrite or completely change how something works, especially if it is not drastically broken in the first place.

The result of this evolution is that understanding what code you need to write in order to query the published content cache is not as trivial as you might assume. Ideally, there would be one single way of accessing an API, however, this is not the case.

Within Umbraco v9, there are a number of different techniques for querying the CMS. The best technique to use within a given scenario will depend on where in the request pipeline the code is being triggered and from what class.

The good news is that all of these different techniques essentially map to the same backend code. All the APIs will get data out of NuCache and return them as `IPublishedContent`. If Umbraco was rewritten from the ground up in .NET Core, I would assume that the API to access the cache would be rewritten to be more straight-forward. As Umbraco has been around for 20 years this is easier said than done. Keeping this point in mind is handy when you first start learning about access content.

UmbracoContext

To access content from the published cache you will need access to an object named Umbraco Context. There are a few nuances and caveats on how you access the Umbraco context.

In the ASP.NET framework, a request was very tightly coupled to an object known as the HTTP Context `UmbracoContext` was based and extended on top of the ASP.NET Frameworks `HttpContext` object.

`UmbracoContext` extended the `HttpContext` object and also added additional Umbraco related properties and operations. A utility called the Umbraco Helper could then be used from within a view or a controller to access Umbraco content via the `UmbracoContext` out of the frontend cache. An example of the code required access content in v8 and below is shown below:

```
var currentPageId =  UmbracoContext.Current.PageId;
Umbraco.TypedContent(currentPageId);
```

View Component Example

This code no longer works in Umbraco 9 for several reasons. The most noticeable one is that it uses a singleton pattern to provide access to the `UmbracoContext`. After establishing a connection to `UmbracoContext` you can access data about the current Umbraco page. This is how the code accesses the current page `Id`. Using the Umbraco Helper, the `Id` is passed into its `TypedContent()` method.

Performing the same type of operation within v9 is very different. First, the reliance within ASP.NET Core on `HttpContext` has changed. As you will see later, `HttpContext` still exists within the framework, however, the way you access it (if you need to) is very different. This has meant that the Umbraco APIs had to be updated to reflect this as well.

Within v9 the `UmbracoContext` has been written against an interface `IUmbracoContext`. If you want to query the CMS to access some content, avoid directly injecting `IUmbracoContext` into a class . Umbraco will automatically register the core APIs that you are meant to use within the dependency injection container. If you tried to inject `IUmbracoContext` as a constructor argument, you would encounter an error similar to this one:

```
Unable to resolve service for type 'Umbraco.Cms.Core.Web
```

This failure to resolve error is thrown because injecting `IUmbracoContext` directly into a class is not the supported way to access the CMS. You will need to access an intermediary API first before you can get access to a fully instantiated `IUmbracoContext` object.

It is possible to access the Umbraco Context object directly within a Umbraco controller without having to use dependency injection. If you create a controller that inherits from `RenderController`, you will have access to a public property called `UmbracoContext` which is of type `IUmbracoContext`:

```
public class HomepageController : RenderController
{
    public override IActionResult Index()
    {
        var content = UmbracoContext.Content.GetByRoute("/");
    }
}
```

Umbraco Context

The `UmbracoContext` exposes a fully functioning object of type `IUmbracoContext`. Using this object will then allow you to access the content cache.

Obviously, you will also need to access Umbraco Context from other areas within your codebase. How do you access the context when you do not have access to a `UmbracoContext` property? If you can not directly, inject `IUmbracoContext`, what do you do?

To follow good practices, when you need access to Umbraco Context, you will still want to access it using dependency injection. This is possible, however, you have to use one of two alternative APIs to gain access to this elusive object. Which API you need to use is further complicated by where in the pipeline the request is being made!

If this is making it sound like Umbraco is complicated to use, relax. For day-to-day development, I will shortly recommend a single approach that you can use in all situations, regardless of where in the codebase you need access. As the aim of this book is to make you a Umbraco master, we need to cover all these different scenarios!

Regardless of which of these two APIs you use to access the context, under the hood, they will both use the `IUmbracoContext` API to access the published content cache. The reason why two APIs exists is a result of how the ASP.NET Core pipeline works.

`UmbracoContext` is built upon the .NET `HttpContext`. The `HttpContext` object relies on the `HttpRequest` in order for it to be cor-

rectly instantiated and fully functional.

If you tried to access certain operations that `IUmbracoContext` provides without the `HttpContext` object being fully cocked, locked and ready to rumble, Umbraco will throw an error. This situation makes sense as all pipelines are based on timing.

If you are solely writing code at the controller level, this timing issue is something that you will never need to consider. Within Umbraco, it is also possible to have code trigger at different stages within the .NET request pipeline.

For example, it is possible to write custom start-up scripts. You could configure some code to trigger when Umbraco first launches on application start. At this stage in the pipeline, the `HttpContext` object may or may not be fully functional.

The takeaway is that depending on what stage within the pipeline code is triggered, will depend on what APIs are available to use. At the very start of a request, the `HttpContext` object will not be fully populated. By the time the request hits a controller, it definitely will be.

If you want to hook into the Umbraco pipeline, you run the risk of code triggering before `IUmbracoContext` is ready to be used. This potential of accessing Umbraco Context prematurely is also true if you want to access the cache from code that lives outside of the normal request pipeline, for example, within a scheduled task.

The important takeaway from this section is that asides from writing the code to call the content cache, you also need to take into consideration the area within the pipeline that the request is made. In certain areas, the `IUmbracoContext` will not be ready by default.

Due to this limitation, there are two different APIs that are available to access Umbraco Content. One assumes everything is fine and the `HttpContext` is fully loaded. The other API allows you to perform a check to ensure the object exists, otherwise it will create one.

IUmbracoContextFactory and IUmbracoContextAccessor

The `IUmbracoContextFactory` is the API that contains the additional validation check. The useful thing to know about this API is that under the hood, `IUmbracoContextFactory` also makes use of the second API, `IUmbracoContextAccessor`

The good news is that it is possible to use dependency injection with both `IUmbracoContextFactory` and `IUmbracoContextAccessor`. Both APIs will ultimately give you access to the same thing. The differentiator is how the Umbraco context is accessed.

Injecting `IUmbracoContextFactory` into a constructor will give you access to an object of type `UmbracoContextFactory`. This object exposes a method called `EnsureUmbracoContext()`. The code within

`EnsureUmbracoContext()` takes into consideration this pipeline dilemma and provides access to Umbraco context in a safe manner. The code to inject and access the context using `IUmbracoContextFactory` is as follows:

```
public class QueryService
{
    private IUmbracoContextFactory _umbracoContextFactory;

    public QueryService(
        IUmbracoContextFactory umbracoContextFactory)
    {
        _umbracoContextFactory = umbracoContextFactory;
    }

    public IPublishedContent GetPage(int id)
    {
        using (var cref = _umbracoContextFactory.EnsureUmbracoContext())
        {
            var cache = cref.UmbracoContext.Content;
            return cache.GetById(id);
        }
    }
}
```

Getting A Page Using Umbraco Context Factory

Within `EnsureUmbracoContext()` a check is made to see if a valid `UmbracoContext` already exists within the current thread. This check is made using `IUmbracoContextAccessor`. If the check returns `true`, the context is returned, job done. If the check returns `false` then `EnsureUmbracoContext()` will instantiate a new object of type `UmbracoContext`.

The differentiator with `IUmbracoContextAccessor` is this check. `IUmbracoContextAccessor` purpose in life is to provide access to the current `UmbracoContext`. If you try to use `IUmbracoContextAccessor` in a non-web request, say a scheduled task, `IUmbracoContextAccessor` will always return null. Using `IUmbracoContextAccessor` within a vanilla MVC controller is fine and dandy. Using `IUmbracoContextAccessor` within a scheduled task will be upsetting.

The code to inject and access the context using `IUmbracoContextAccessor` is shown below:

```
public class ExampleClass
{
    private IUmbracoContextAccessor _context;

    public SearchService(IUmbracoContextAccessor context)
    {
        _context = context;
    }


    public IPublishedContent GetUmbracoObject(int id)
    {
        var context = _context.GetRequiredUmbracoContext();
        return context.Content.GetById(id);
    }
}
```

Getting A Page Using Umbraco Context Accessor

This code uses `GetRequiredUmbracoContext()` to get a reference to a `UmbracoContext` object.

IPublishedContentQuery

There is also another API you should be aware of IPublishedContentQuery. As the name implies `IPublishedContentQuery` is a query-specific API for accessing items within the published content cache. This API exposes methods like `Content(int id)`, `Search()`, `Media()` and `ContentAtRoot()`.

The `IPublishedContentQuery` will give you direct access to the published content cache without any checks at all. You can use dependency injection with this service, however, you can only use `IPublishedContentQuery` within the stages of the pipeline where the `UmbracoContext` exists.

This essentially means that you should only use `IPublishedContentQuery` at a controller or view level. Within the controller initialization stage, you will always have access to `UmbracoContext`. The code to access some content using `IPublishedContentQuery` is shown below:

```
public class ExampleClass
{
    private IPublishedContentQuery _publishedContentQuery;

    public SearchService(IPublishedContentQuery publishedContentQuery)
    {
        _publishedContentQuery = publishedContentQuery;
    }


    public IPublishedContent GetUmbracoObject(int id)
    {
        var context = _publishedContentQuery.Content(id);
    }
}
```

Getting A Page Using Published Content Query

My advice around the usage of the `IPublishedContentQuery` API is to avoid using it unless you want to perform a search against many pages rather than querying for a single page. Searching is a topic in itself and will be covered later on in this chapter.

The output of all three of the code snippets listed within this section results in the same data being returned. Do not forget you can also access the `UmbracoContext` if you are within a Umbraco controller via the `UmbracoContext` base property.

To re-cap, yes, there are four different ways of writing code to do the same operation! This is why I started this section by talking about Umbraco's evolution. If you were to recreate a CMS from the ground up, you would likely only create one API for getting content from the CMS, however, as Umbraco is well established it has many ways of doing the same thing!

After first learning about all these different APIs and the different rules of when they should and should not be used, it is hard not to feel further confused, I get you. My mental capacity does not have the will or energy to remember every nuance and rule for every API for every framework and programming language that I work with.

A very easy rule of thumb to follow that will work in every situation is to always use `IUmbracoContextFactory`. Following this rule means that you will end up writing a little bit more code here and there, however, the trade-off is that your codebase will be consistent. As an added bonus you will only need to remember how to write code for one API!

Umbraco Context Deep Dive

Using one of the techniques described in the last section, you should be able to access `UmbracoContext` from anywhere within your codebase. In this section, we will review exactly what `UmbracoContext` has to offer.

As we now know, when a new request is made to your application, a new `HttpContext` and subsequent `UmbracoContext` object will be instantiated by the framework. Through `UmbracoContext` you will get access to several handy Umbraco related properties and operations.

The property that you will access most frequently is the `Content` property. The `Content` property is the property that provides access to NuCache and all the contents of the published content cache. Whenever you want to query Umbraco for information about a page not related to the current request this is the property that you will use.

The `Content` property is of type `IPublishedContentCache`. `IPublishedContentCache` also implements from an interface called `IPublishedCache`. Its the methods and properties exposed by these two interfaces that provide you with the gateway to the content cache.

Do not try to access `IPublishedContentCache` or `IPublishedCache` directly. You can prove this by trying to directly inject `IPublishedContentCache` into a constructor. Doing so will result in an 'Unable to resolve service' exception.

Asides from the `Content` property you will also have access to these additional things from `UmbracoContext`:

Media: Whenever an image, file, or allowed media type is uploaded into Umbraco via the media library, a reference to that item is also added within the frontend cache. The `Media` property which is of type `IPublishedMediaCache`, provides access to these items.

The `Media` property will likely be your second most used property from `UmbracoContext`. Depending on which strategy you pick to manage global settings, it is less likely you will need to access media using this property. When content modelling, I tend to model global images and assets within my global settings.

The choice of how you model your global settings is up to you, however, if you opt to add global assets within the area you use to model global settings, the need to query for content using the `Media` property is pretty rare.

Just like `IPublishedContentCache`, you should not try to directly access `IPublishedMediaCache` yourself. Injecting it as a constructor argument will also result in an 'Unable to resolve service' exception being thrown.

Asides from the `Content` and `Media` properties, I tend to very rarely use the other properties exposed from `UmbracoContent`. I would not recommend spending time memorizing everything provided by `UmbracoContext`, however, for completeness here is a complete list:

ForcedPreview(bool preview): When creating and editing content from within the Umbraco backend, it is possible to also preview it. The mechanism of how Umbraco differentiates between a normal request and a preview request is a flag stored within `UmbracoContext`.

`ForcedPreview()` can be used to force toggle the current request into a preview request. Force enabling preview mode can be useful for custom backend development.

If you want to write a custom backend screen and you want to get data about yet to be published content, forcing a request into `preview` mode will return additional information that is not otherwise available. An example of some additional data is the items `Segment`.

InPreviewMode: A `bool` flag that details whether the current request is set to preview mode or not.

ObjectCreated: A `DateTime` property detailing when the current `UmbracoContext` object was instantiated.

OriginalRequestUrl: This property returns the original requested URL, respecting the original text casing.

CleanedUmbracoUrl: This property returns a cleansed Url version of the requested Url. `CleanedUmbracoUrl` will return the current request Url in lowercase, with no trailing slashes, and no trailing extensions.

IsDebug: A `bool` flag that details whether the request has been set to debug mode.

PublishedSnapshot: Gets a snap-shot point in in time of the pages, media, and member data within the CMS

PublishedSnapshot: Returns a point in time snapshot of NuCache. PublishedSnapshot is of type `IPublishedSnapshot` and returns a snap-shot in time view of all the pages, media, and member data within the frontend cache.

This property can also be used to access properties such as `Content`, `Media`, `Domains`, and `Members`. Do not try to access cached items via `PublishedSnapshot` directly. If you want to access data from the cache, use the properties exposed from `UmbracoContext`.

'PublishedSnapshot' can potentially be useful for unit testing the Umbraco context, however, in reality, this property is mainly used by the core code and you can forget about it.

Domains: As you will learn later, you can build multi-language websites and micro-sites within a single Umbraco instance.

When working with multiple languages and multiple micro-sites, you will likely need to have different website domains and/or sub-domains resolve to different areas within the Umbraco page tree.

How to configure Umbraco with different domains is covered within the multi-language website section. The main takeaway is that the domain can differ between requests.

The `Domains` property can be used to access all the domains that have been added by content editors inside of the CMS. You can access this data using `GetAll()`. You can also query this list for the configuration of a specific page using `GetAssigned()`.

PublishedRequest: The `PublishedRequest` property provides lots of useful information about the current request, like `Culture`, `Domain`, `Headers`, `ResponseCodeStatus`, and `Template`.

In general, you will work with the `Content` property very frequently while building a project within Umbraco. The other properties can be useful in specific situations, however, the need to access these properties will usually be down to specific edge cases.

You can learn more about `UmbracoContext` by reading the source code for the class which is on GitHub here

Umbraco Helper

Even though we have covered four potential ways of accessing the Umbraco published items cache using `UmbracoContext`, amazingly there is still another way of accessing this cache to cover. This final method is called `UmbracoHelper` and it can also be used to access the published content cache.

`UmbracoHelper` is definitely not a new API within the Umbraco world. I started using Umbraco to build websites when v6 was just released. `UmbracoHelper` existed way back then and it is still here today.

`UmbracoHelper` is a utility class whose sole purpose is to provide access to the Umbraco frontend cache. `UmbracoHelper` exposes operation like `Content()`, `ContentAtRoot()`, `ContentAtXPath()`, `MediaAtRoot()`, `RenderTemplateAsync()`, and `Media()`.

Under the hood, `UmbracoHelper` is just a wrapper on top of `UmbracoContext`. This means `UmbracoContext` needs to exist for `UmbracoHelper` to work. Just like `IPublishedContentQuery`, you should only use `UmbracoHelper` within the controller and view layers.

Sadly the `UmbracoHelper` has still not been written against an interface in v9 or v10. You can use dependency injection with `UmbracoHelper`, however, you will be injecting the `UmbracoHelper` class directly.

In terms of software architecture, having to inject a type into a constructor is not as desirable as injecting an interface. Injecting a type rather than an interface provides less flexibility. When working with type injection, you will not be able to mock as much when writing unit tests. You will also not be able to apply SOLID principles and adhere to the open/closed principle quite so well.

In previous versions of the CMS, it was much harder to access the `UmbracoContext` than it is now. Before route hijacking, it was more common for Umbraco developers to write code directly in the view layer.

Now we have route hijacking and dependency injection out of the box, there are no technical blockers preventing you from fully adhering to the MVC paradigm. With `IContextFactory` and `IContentAccessor` you have two flexible methods for accessing the `UmbracoContext` anywhere in the codebase. This means that in v9, `UmbracoHelper` class is less useful than it used to be.

The code to use `UmbracoHelper` is intuitive. Inject it into a constructor and use it:

```
public class ExampleClass
{
    private readonly UmbracoHelper _helper;

    public ExampleClass(UmbracoHelper helper) =>
        _helper = helper;

    public void MyMethod()
    {
        var rootContent = helper.ContentAtRoot();
    }
}
```

Umbraco Helper Example

The Umbraco website still calls UmbracoHelper as the unified way to work with published content, however, until `UmbracoHelper` is written to an interface I would avoid using it.

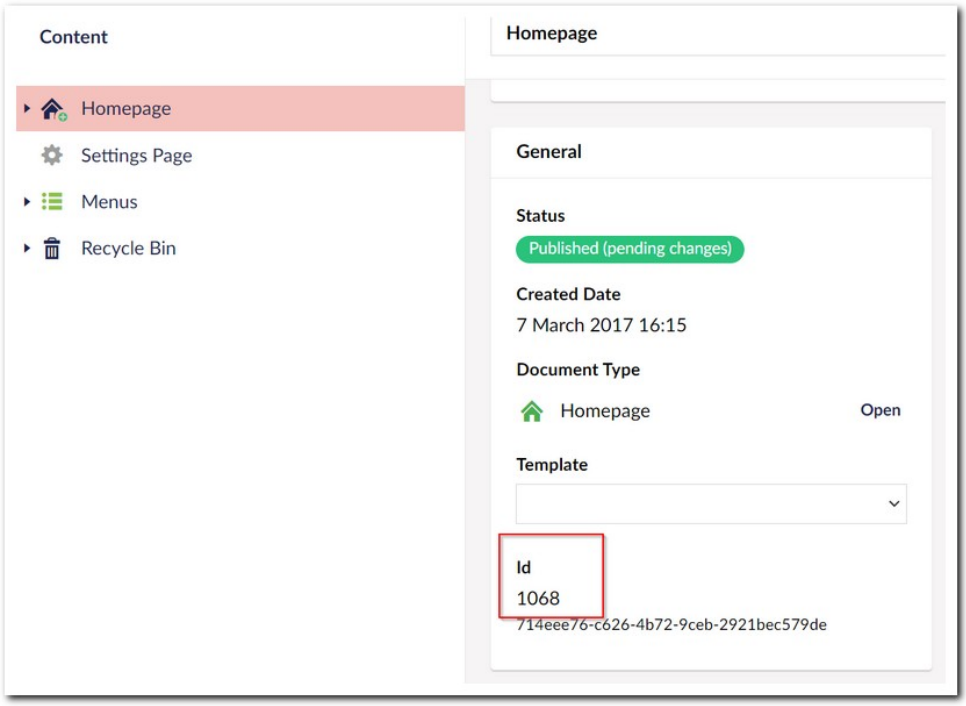
Querying Umbraco For Published Content

After covering the multitude of ways of accessing the published content cache, let us get to the good bit, querying the CMS for stuff! In this section, you will learn about the operations that can be performed against the cache so you can render whatever content (or media) that makes you happy.

Questions that will be answered within this section include, how do we access a particular page, how do we access the homepage, how do access global settings, how do you access content within a multi-language setup

and how do you query the CMS by content type? `IPublishedContent` exposes a number of handy operations, let us look at them in detail:

GetById: This method will allow you to query for any published content page within Umbraco using its page `Id`. The `Id` for any content item can be found within the pages editing screen within the `Info` tab:



Getting the ID From Within The CMS

You can also access the `Id` from the internal URL within the editor. Whenever you edit a page within the CMS, if you look within the browsers Url bar, the Url will look similar to this:

`https://www.mywebsite.com/umbraco/#/content/content/edit/`

In this example, `1068` is the `Id`. Take this value and then pass it into `GetById()` to get a corresponding item:

```
var content = UmbracoContext.Content.GetById(4);
```

GetById()

My recommendation is to favour this method whenever possible. You will find that its much safer to query a page by its `Id` rather than its `Url`. If a

content editor renames a page within the CMS or moves the location of the page within the page tree, the `Url` will change, however, the `Id` property will stay the same.

GetByRoute: This method will allow you to query the CMS based on a Url segment. Pass in the segment for the page and get a reference to the object in `IPublishedContent` format:

```
var homePage = UmbracoContext.Content.GetByRoute("/");
```

GetByRoute()

I find that `GetByRoute()` is the easiest way to query for the homepage object within Umbraco. The `Url` for the homepage will never change. Every websites root Url will always be `/`. Code that references the root domain can never be accidentally broken by a content editor meaning querying the CMS for the homepage using `GetByRoute()` will always work!

GetAtRoot: This method will return all the pages underneath the root item within the Umbraco page tree. If the content editor has only added a single homepage item under the root node, this is all that will be returned within a list. If your Umbraco instance hosts multiple websites, or you have added your global settings within the settings document type that has been created under the root node these items will also be returned with this query.

The code to perform this query is shown below:

```
var rootChildren = UmbracoContext.Content.GetAtRoot();
```

GetAtRoot()

If you opted to content model your global settings within settings specific document types, this is the query that I tend to favour to access global settings. I prefer querying rather than hard-coding Ids to access global settings for added flexibility, however, this is definitely a personal preference and not a best practice you need to follow.

GetContentType: This method will return all pages created within the CMS using a certain document type. Pass in the document-type alias to get all the corresponding pages:

```
var landingPages = UmbracoContext.Content.GetContentType("alias");
```

GetContentType()

Whenever you need to access a document-type alias there is a handy tip that is worth keeping in mind. Any class generated by the Umbraco models builder also exposes its alias via a static property called `ModelTypeAlias`:

```
var homepageAlias = Home.ModelTypeAlias;
```

Accessing The Homepage Alias Using Models Builder

Whenever you need to reference an alias in code, using this property will result in a slightly safer architecture compared to hard-coding the alias value directly within your source code.

GetSingleByXPath(), GetByXPath() and CreateNavigator(): There is an alternative approach for querying the published cache, using XPath queries. Being able to query the cache using XML might be a surprising option, however, this feature is something that has been around within Umbraco for a long time.

The reason why we have XML based queries is again based on history. Before NuCache was released, the Umbraco cache was based on an XML file called `umbraco.config`. This config file contained an XML representation of everything added within the cache. Being able to query the cache using an XPath query made a lot of sense back then, especially as you could see the XML structure easily by opening `umbraco.config` within a text editor. Even though NuCache replaced `umbraco.config`, it is still possible to use XPath to query NuCache.

There are three XPath based methods exposed by `IPublishedCache`. The main two being `GetSingleByXPath()` and `GetByXPath()`. As you can deduce by the naming convention, one approach is used to get an individual item from the cache, the other, to get many. The input parameter for both APIs is the same, a query. An example of how to query using `GetSingleByXPath()` is shown below:

```
var content = UmbracoContext.Content.GetSingleByXPath("//{alias}");
```

GetByRoute()

The other XQuery based option is `CreateNavigator()`. `CreateNavigator()` takes a single optional parameter, `preview`, and returns an XPath navigator object. A XPathNavigator object can then be used to navigate or the content caches using its XML representation, e.g. the contents of NuCache:

```
var navigator = UmbracoContext.Content.CreateNavigator();
navigator.Select("descendant::node()[@id='{0}']");
```

CreateNavigator()

Querying the cache using XPath is an Umbraco ability that has been around for ages. There might be a handful of old-school Umbraco developers who swear by querying the cache using XQuery, however, personally, I recommend that you avoid these operations.

If you tried to Google for information on any of these methods, you will quickly see there is limited information or examples of how these methods are used online. This lack of searchable content is always a sign of how infrequently something is used in production.

Querying the cache using XPath is a Umbraco ability that has been around for ages. There might be a handful of old-school Umbraco developers who swear by querying the cache using XQuery, however, personally, I recommend that you avoid these operations.

If you try to Google for information on any of these methods, you will quickly see that there are limited results and examples of how these

methods are used online. This lack of searchable content is always a sign of how infrequently something is used in production.

If you want to query Umbraco for a particular page, I recommend you favour one of the non-XML related methods, like query by Id, route or content type.

GetRouteById: The `GetRouteById()` methods, takes a content `Id` and returns the items segment/url as a `string`. For day-to-day webpage creation, I have yet to find a really good use case for needing to use this method.

The reason for this is that there are easier ways to get the Url from an item within the published cache. Umbraco ships with an extension method that can be used with any object of type `IPublishedContent` called `FriendlyPublishedContentExtensions`, that returns an items Url.

`FriendlyPublishedContentExtensions` is found within the `Umbraco.Extensions` namespace. `FriendlyPublishedContentExtensions` contains a number of useful extension methods, including a method called `Url()`. If you want to get a `Url` for a page, it is much more convenient to just work with `IPublishedContent` and the `FriendlyPublishedContentExtensions` extension. If you are determined to use this method, the code to use it is below:

```
var urlSegment = UmbracoContext.Content.GetRouteById(CurrentPage.Id);
```

GetRouteById()

I feel `GetRouteById()` is more used by the Core CMS itself rather than CMS implementors. Unless you bump into an edge case where `Url()` on `IPublishedContent` does not work, avoid using this method.

IPublishContent

After you can access an item of type `IPublishedContent` you will typically want to do one of two things. Get a bit of data from the content item, or, get re-query the cache in order to get content related to that page. Common examples when you need to perform this type of operation are render menu-items when you need ot get child pages and rendering a breadcrumb when you need to get all the descendants of an item. In this section, we will do a deep-dive into all these properties and methods.

Id: This property is simple enough, the unique identifier of the content item. You will use this `Id` value a lot so get used to it!

Name: This property maps to the mandatory name property within the CMS. Often in content modelling the title will be used as the pages `<h1>` tag. It is impossible to create an item within Umbraco without a name, so you can use this property with confidence.

Parent: Returns a reference to the items parent node in `IPublishedContent` format. The parent of your homepage will be the root node. The parent of the root node will be null.

Path: Returns the path of the item in relation to the page tree within the CMS. Do no confuse this with the pages external Url!

Level: Returns the internal tree depth of the item within the page tree.

SortOrder: All content pages within the CMS will have a parent. Items under a parent can be ordered however the editor desires. `SortOrder` returns the internal sort order of the item in comparison to its siblings within the page tree.

TemplateId: Do you remember that within Umbraco, a MVC view is reference to as a templates. When a content editor creates a document-type, Umbraco can also create a corresponding view/template/cshtml file for you. This view/template can be referenced inside the CMS. This property returns the view `Id`. This is handy if you want to dynamically change the template associated against a page on render. This type of scenario is very rare, however, you have that ability if needed!

ItemType: Returns an enum of type `PublishedItemType` that explains the content type. This value can be either `Unknown`, `Element`, `Content`, `Media`, or `Member` .

CreateDate: Gets the date the item was created within the CMS.

WriterId: Within Umbraco each content editor account has an internal Id. The writer Id relates to the content editor who last updated the content item.

CreatorId: The create Id relates to the content editor who created the item.

Cultures: Within a multi-language website, a page might be translated into a number of different languages. This property return a list of all of those translations. Multi-language will be covered in detail later.

UpdateDate: Returns the date the page was last updated, typically this is the last published date. This date does not take into consideration the culture value. When any language version of a page is made, this value is updated.

Children: Returns a list of all the related child pages that the content editors have created underneath the page within the CMS. The items in this list are also all of type `IPublishedContent`.

ChildrenForAllCultures: Returns a list of all the related child pages based on culture. This will return references regardless of whether they have been translated or not.

IsDraft(): Returns a boolean indicating if the current content has been published. This is mainly used by the preview to determine if it needs to display the published version, or, if it needs to render an in progress version of the page.

IsPublished(string culture = null): Returns a boolean indicating if the item has been published. When using the APIs normally this will always be `true`.

FriendlyPublishedContentExtensions

Asides from the default method, it is also possible to add some additional useful methods to any object of type `IPublishedContent` that you are working with via `FriendlyPublishedContentExtensions`. `FriendlyPublishedContentExtensions` contains lots methods, most of which are overloads. There are overloads for `Children`, `Name`, and `UrlSegment` that take into consideration the culture. Asides from these handy overloads you get a lot of useful node traversal methods like:

Descendants(): Returns a list of all the related children, grandchildren, great grandchildren, etc.. to the current page. The list is of type `IPublishedContent`.

DescendantsOrSelfOfType(): This does the same as `Descendants()`, however, it also returns a reference to the current page as well.

Siblings(): Returns a list of all the current items siblings.

FirstChild(): Return an object of type `IPublishedContent` based on the first child of the current page.

As previously mentioned you also get access to `Url`

GetTemplateAlias(): Returns the pages template alias. In plain talk, the name of the view

IsAllowedTemplate(): It is possible to associate multiple templates to a page. This method allows you to validate that a given template can be used with a page.

SearchDescendants() and SearchChildren(): These items allow you to perform a search, rather than a query for content. Search will be covered in more detail in the next section.

Searching For Content

In order for a result(s) to be returned using one of the APIs like `IPublishedCache`, an exact match needs to be made. Some bit of data

contained within the potential result set needs to exactly match the query criteria.

Querying the CMS for a particular page is extremely useful when you know exactly what page you need to access. Often, you will not always want to perform exact match look-ups. Instead, you will need to perform an inexact look-up in order to get data from Umbraco. In these situations, you will need to search the entirety of the content tree based on a certain term, or, keyword.

When performing a search, a relevancy score will be calculated between the search term and the potential result set. Higher relevancy results will be returned first, and lower relevancy results last. The code to create an API that performs a search is very different compared to the code required to simply state "match" or "not a match".

As the underlining criteria between querying and searching are very different, Umbraco provides different APIs for querying and searching. Querying the published content tree is done within NuCache. Searching for content is typically done using Examine.

Examine can be thought of as an Umbraco powered search engine. Technically, under the hood Examine is a bespoke search wrapper. Examine wraps a very popular search engine called Lucene. Lucene works by spidering content and then providing relevancy look-up capabilities.

The reason why we need Examine in Umbraco is because Lucene is not a C# or Umbraco specific search engine. Examine is needed to provide a gateway between the low-level query language of the Lucene engine and code that we can work with in a C# project. Examine allows us as developers to easily work with Lucene using C# and get results typed to `IPublishedContent`.

Items are automatically spidered by Lucene whenever a content editor creates or edits a page within the CMS. The Umbraco search APIs, Examine and Lucene are all included as part of the core CMS. This means that you can perform searches within Umbraco easily without having any additional setup tasks to worry about!

Examine Overview

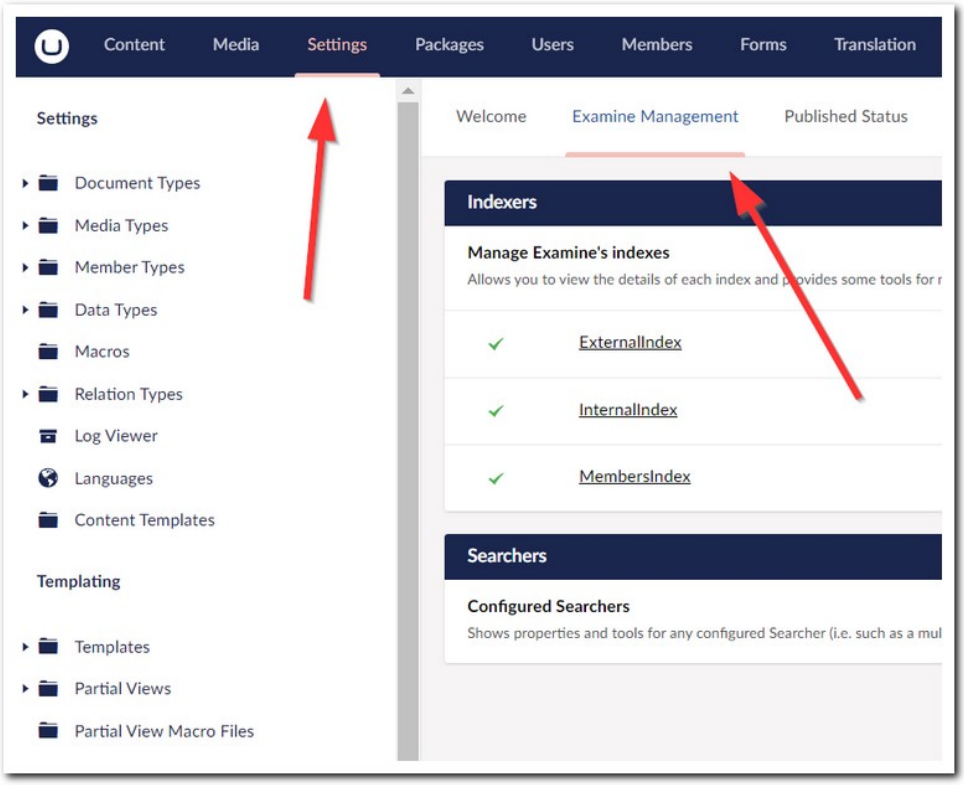
After Lucene successfully spiders some content, it stores the results within an index file. In previous versions of Umbraco, the index files could be found within the website's webroot folder, within a sub-folder called `App_data`.

The `App_data` folder is a special .NET Framework folder, however, it is not needed within a .NET Core application. This means that the location of the index files has changed within a Umbraco 9 website. The indexes can now be found within a new location, `umbraco\Data\TEMP\ExamineIndexes`.

The end search index files created by Lucene are exactly the same as in previous versions of Umbraco. If you want to view the contents of these

index files, you can use a handy tool called Luke For Lucence. Luke For Lucence will allow you to inspect the results and perform queries against the index yourself.

A second and easier way of querying the contents of the Umbraco index can be found within the CMS. Umbraco also ships with a handy search tool that is very useful for debugging search-related issues. You can find this search tool within the `Settings` section within a tab called `Examine Management`:



Accessing Examine From The Backend

As you can see from this screenshot, Umbraco ships with three different search indexes. Each index contains slightly different data. When you perform a search within Examine, you need to specify the index that you want to perform the search against.

It is also possible to extend Examine and define your own custom search indexes. I am not covering custom indexes here, however, there is a link in the extra reading section to help you learn more.

Out-of-the-box, the three included indexes are:

ExternalIndex: This index contains all public-facing content and media. Typically `ExternalIndex` is the index you will be using the most. You can use this index as the data source for any site search.

InternalIndex: This index contains all published and non-published content and media within the CMS. Whenever you use the CMS to look up content, `InternalIndex` is the index used to power the internal CMS search.

As `InternalIndex` includes pages that have not been published, you should avoid using `InternalIndex` as the data source for any public-facing pages. `InternalIndex` is more useful if you need to build any custom backend screens with search.

MembersIndex: This index contains a list of all the members defined within the CMS. Remember a member is used for public-facing member areas or newsletters. This list does not contain content editors. `MembersIndex` is handy when you need to build a membership site. The member related APIs will be covered in a later chapter.

Having three out of the box indexes is handy. Regardless of your requirements, you have an option to search Umbraco for that data easily!

IPublishedContentQuery and IExamineManager

In order to write code that performs a search within Umbraco, you can use one of two APIs, `IExamineManager`, or, `IPublishedContentQuery`.

Both `IPublishedContentQuery` and `IExamineManager` will allow you to search Umbraco using Lucene. Of the two, `IExamineManager` is definitely the more feature-rich API.

IPublishedContentQuery: `IPublishedContentQuery` combines both query-based methods and search-based methods into a single API. `IPublishedContentQuery` provides access to query based methods like `Content(object id)` and `ContentAtRoot()`. `IPublishedContentQuery` also contains a search based operation called `Search()`.

According to the Umbraco website, the purpose of `IPublishedContentQuery` is to allow querying of strongly-typed content in templates. Writing code within a template/view is an MVC anti-pattern. This is why my recommendation is to avoid this API whenever you use route-hijacking within your project.

The `Search()` method within `IPublishedContentQuery` allows you to perform a search against the `ExternalIndex`. The result of `Search()` is a list of items of type `PublishedSearchResult`. `PublishedSearchResult` contains two properties the result and a search score.

The `IPublishedContentQuery.Search()` method has three overrides. The simplest version takes a single argument, search term, and returns any corresponding results:

```
public HomepageController(
        IPublishedContentQuery publishedContentQuery
        )
{
    publishedContentQuery.Search("searchTerm");
}
```

Simple Search

The second overload allows for searching with pagination. Asides from the search term, the second option allows for two additional arguments to be provided, `Skip` and `Take`. `Skip` and `Take` are common pagination names. `Skip` defines the current page number and `Take` defines the page size:

```
public HomepageController(IPublishedContentQuery publishedContentQuery)
{
    int skip = 1;
    int take = 10;

    publishedContentQuery.Search("searchTerm", skip, take, out long totalRecords);
}
```

Simple Search

The final overload is more advanced and allow a query object of type `IQueryExecutor` to be passed into it. I am not including the code for this final method for a reason. In order to use it, you need to also use `IExamineManager`. `IExamineManager` also allows for exactly the same capability, so if you can do everything within a single API, why would you ever use two?

IExamineManager: `IExamineManager` provides everything you need to perform simple and complex searches against any index. In general, my recommendation would be to write any search related code within a custom service and favour using `IExamineManager`. `IExamineManager` will give you more power to change the search behaviour later on if needed.

You can use dependency injection with `IExamineManager`, simply pass it into a constructor as an argument:

```
public class MyController : RenderMvcController
{
    private readonly IExamineManager _examineManager;

    public MyController(IExamineManager examineManager)
    {
        _examineManager = examineManager;
    }
}
```

IExamineManager() Example

To perform a search against the external index using a search term, can be done like this:

```
var indexName = "ExternalIndex";
var searchTerm = "search term";

if (_examineManager.TryGetIndex(indexName, out var index))
{
    var qry = index.Searcher.CreateQuery("content").Field("nodeName", searchTerm);
    var results = qry.Execute().OrderByDescending(x => x.Score);
}
```

Content search example using IExamineManager

As you can see the code to perform a search using `IExamineManager` is a little bit more complicated compared to `IPublishedContentQuery`. The trade-off is a lot more power and flexibility. On a typical project I will put all my search related code into a single custom class called `SearchServer` and then use that single class to allow for searches to be made anywhere in my code base. The nice thing about using `IExamineManager` is that flexibility to change things as we go along.

To get going within `IExamineManager`you will need to decide which index you want to query against. You do this using `TryGetIndex`. Out of the box Umbraco provides a handy `Constants` class, you can use to access all the default index names `Constants.UmbracoIndexes.ExternalIndexName`

For 90% of your use cases, you will want to query the `ExternalIndex` for content. Once you have From the index object, using the `Searcher` property, you can define a search query using `CreateQuery` . You can make this query as simple or complex as you want. The result of this

query is an object of type `IQuery`. This method exposes a method called `Execute()`. Using `Execute()` you can then get a list of all the results.

Search parameters

When building a query object you have lots of options. Often knowing how to build the query is the hardest part. In this section, I'll run through some handy code samples that will get you on your way.

Filter By Document-Types: Searching for pages filtered by a certain document-type is possible using the `Field()` method. `Field()` takes a single value, the property alias that you want to search on. The document-type alias is stored in a field called `__NodeTypeAlias`. Simply, query that field with the document type alias, like this:

```
var qry = index.Searcher.CreateQuery("content")
                        .Field("nodeName", "searchTerm")
                        .And().Field("__NodeTypeAlias", "aliasToSearch");
```

Filter By Document Type

Filter By Content Type: The external index contains references to all published content and media. Sometimes you will want to filter the results so only content or media is returned. Filtering by content is again done using the `Field()` method. The content type data is stored within a field called `__IndexType`. The query language comes with the option of appending additional query praters using a full set of conditional operators. You get access to methods like `And()`, `Or()`, and `Not()` so you can chain commands together and create as complex a query as you need.

By using the `Not()` operator we can exclude results by a certain field. TO filter by content type we can filter by either `media` or `content`.

```
var qry = index.Searcher
                .CreateQuery("content")
                .Field("nodeName", "searchTerm")
                .Not().Field("__IndexType", "media");
```

Filter By Content Type

Sorting Results: Search results can be filtered using `SortableField` and `OrderByDescending()` or `OrderBy()`. Create a new `SortableField`

object, define the field you want to sort on and pass it into one of the sorting filters`:

```
var field = new SortableField("myPropertyAlias");

var qry = index.Searcher.CreateQuery("content").Field("nodeName", "searchTerm");
qry.OrderByDescending(field);

var results = qry.Execute();
```

Search By Document Types

Targeting Multiple Fields From A Search: When it comes to building a query that needs to target multiple fields, the recommended way to build that query is to chain all the difference conditions separately.

It is possible to target multiple fields within a single command with either `Group()`, `GroupedOr()`, `GroupedAnd()`, or `GroupedNot()`. together

As Examine provides so many different types of operators, it is possible to query the index for the same data in multiple different ways. I would not say that there are defacto best practices when it comes to search. When setting up a search, I tend to do a trail by fire. Try out a few things are see what works best. I recommend you do the same, test things out and use the combination that works for you needs.

For example, excluding certain fields and values from your results can be done in several ways. You could chain the `Not()` operator and the `GroupedOr()` operator after you use `Field()`, you could use the `GroupedNot()` operator, of you could simply chain `Not()` and `Field()` as many times as you need.

In the example below we two fields are excluded `umbracoNaviHide` and `__IndexType`:

```
var fieldsToExclude = "umbracoNaviHide";

var query = index.Searcher
    .CreateQuery(null).GroupedNot(fieldsToExclude.Split(','), "searchTerm");
```

Example of GroupedNot()

PDF Content: There used to a very handy Examine extension that worked and spidered PDF content. Unfortunately, at the time of writing that plugin has yet to be updated

Content Service

The last key API to cover within this chapter is the `IContentService`. As discussed at the start of this chapter, if you need to work with some content at a CMS level, you can not use the published cache or the search indexes. Some examples of what I consider CMS level tasks include accessing unpublished content, working with blueprints, reading content version history or wanting to update content in code.

If you want to perform these types of operations on content, you will need to use the `IContentService` API. The big disclaimer when using this API is performance. Each operation provided by `IContentService` will make a database call.

In terms of best practices use this API sparingly. Due to the performance impact `IContentService` will cause, you should not call the `IContentService` API from public-facing areas on your website. I recommend aiming to avoid using `IContentService` unless you are writing a backend plug-in within the CMS.

The `IContentService` will return all content as type `IContent`. `IContent` exposes slightly different properties and operations compared to `IPublishedContent`. We will cover these differences shortly.

To access `IContentService` in code, you can use dependency injection and inject it as an argument into a constructor:

```
public class QueryService
{
    private IContentService _contentService;

    public SearchService(
        IContentService contentService)
    {
        _contentService = contentService;
    }

    public IContent GetPage(int id)
    {
        return _contentService.GetById(id);
    }
}
```

Accessing IContentService

I think one of the reasons why it is so easy for Umbraco newbies to get mixed up between the published content APIs and `IContentService` is that they both have a lot of similarities. At a basic level, they expose a lot of the the same types of operations, these include:

- `GetById()`
- `GetByIds()`
- `GetParent()`
- `GetAncestors()`
- `GetRootContent()`

The difference with `IContentService` is that you also get access to some more fine grain operations. As `IContentService` exposes quite a lot of operations, in this section I will only cover the most used operations.

GetVersions(): Whenever a content editor creates a new page within Umbraco, behind the scenes an entry is made within the database. If any subsequent changes are made to that item, the original changes are never deleted. Instead, any additional updates and changes are saved as a new version.

Being able to access the complete version history is useful for lots of companies. For some sectors, like finance, it's a legal requirement to be able to prove what a website promised on a certain date.

`GetVersions()` returns a list of all the versions of a bit of content. Pass in the content `Id` that you are interested in and get the entire content editing history for that item back.

Save(): This operation allows you to pass in an item of `IContent` so that it can be persisted to the database. There is also an additional `SaveAndPublish()` method that does exactly what it says on the tin!

Unpublish(): In terms of content management, not all content published onto a website will live forever. Using the `Unpublish()` method on an item will revert it from the website. Unpublished content is only hidden from the website, however, the item will still be available within the CMS. Any incoming requests to an unpublished page will result in a 404. The code to unpublish a bit of content is shown below:

```
IContent hompage = // Get item
var result = contentService.Unpublish(hompage);
```

Unpublish()

If you are working within a multi-language website set-up, you will also need to consider the `culture` that you want to un-publish as well.

GetContentScheduleByContentId(): This method will only be available within Umbraco 10 and upwards. If a bit of content is scheduled for re-lease, this method will allow you to get all its associated publishing dates. This method is new to Umbraco 10 and it allows for both read and write access. The code to get the related dates is shown below:

```
public class MyClass
{
    private readonly IContentService _servicce;

    public MyClass(IContentService servicce)
    {

        var documentTypeId = 8;
        var scheduledDate = _servicce.GetContentScheduleByContentId(documentTypeId)
                                     .FullSchedule
                                     .FirstOrDefault(x => x.Action ==
ContentScheduleAction.Release);
    }
}
```

GetContentScheduleByContentId()

Delete(): The delete operation completely removes the item from the Umbraco database. Be careful when using this. Make sure you have a database backup!

DeleteOfTypes(): This method is even more dangerous than `Delete()`. `DeleteOfTypes()` deletes all content based on an document type alias. Use this method with extreme caution!

DeleteVersions(): This method will purge all the data about a specific version from the database. This can be useful if you want to keep your database size slim.

MoveToRecycleBin(): This is a safer method to use compared to `Delete()`. Instead of completely blatting the content from the database, this operation will move the item to the Umbraco recycle bin. Moving an item to the bin will remove the item from the content tree while still allowing an editor to revert the decision later on.

EmptyRecycleBin(): Purges all the content contained within the recycle bin from the database.

EmptyRecycleBin()

Create(): The final CRUD (create, read, update, and delete) operation. As well as editing and deleting, you can also create content in code. Pass in an item of `IContent` and job done!

GetPermissions(): Returns a list containing all the permission groups that have been assigned to the content. There is also a corresponding `SetPermissions()` method if you want to change access levels in code.

```
IContent examplePage = // Get item
var permissionCollection = contentService.GetPermissions(examplePage);

foreach(var collection in permissionCollection)
{
    foreach(var permissionGroup in collection.AssignedPermissions)
    {
        // Access to permissions
    }
}
```

GetPermissions()

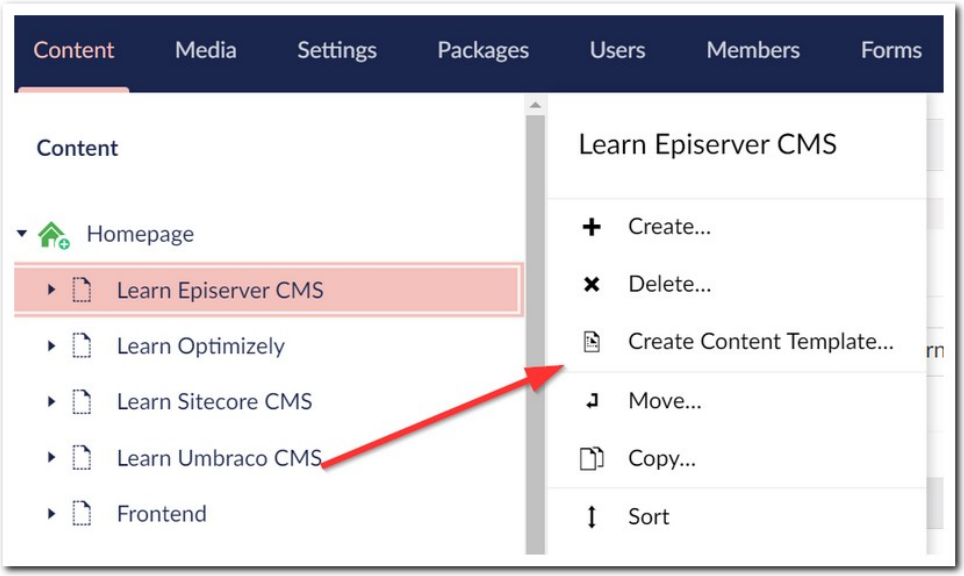
Move() and Copy(): `Move()` will allow you to change the location of some content within the Umbraco page tree. `Copy()` will duplicate an item.

Count(): Counts the number of items that have been created with a specified document type. Pass in a document type alias, and get the count returned.

GetContentForExpiration(): Pass in a date range and gets a list containing all the content that is close to expiring.

Another very handy content enditing time saver that we have yet to cover is the concept of blueprints. When content editors are creating pages within the CMS, they may often need to create pages that are very similar. Same images, meta-data, etc…

Instead of them having to fill out every field over and over again, Umbraco introduced the concept of a Blueprint. A blueprint can be thought of as a reusable content template. Within the CMS if you click on any page, you will have the option to `Create Content Template`.



Content Template = Blueprint

This operation will in essence clone that page and all its content so that it can be re-used as a template. With a blueprint set-up, life is a little bit easier for the content editors. It is also possible to do all the standard CRUD operations on a blueprint. All these blueprint powered CRUD operations are also exposed by `IContentService`. As well as all the operations listed above, you will also find these blueprint related ones:

- `GetBlueprintById()`
- `SaveBlueprint()`
- `DeleteBlueprint()`

IContent

Any item returned by `IContentService` will be of type `IContent`. `IContent` exposes a completely different set of properties and operations compared to `IPublishedContent`.

It is highly unlikely that you will ever need to use a majority of these properties and operations, so rather than going over lots of uninteresting properties, we will focus on the more useful and interesting ones. For a complete list of operations I recommend checking out the developer docs.

As you would expect when working with items of type `IContent`, you will get access to all the properties that you would expect, including:

- `Name`
- `Id`
- `CreatorId`
- `ParentId`
- `Level`
- `SortOrder`

One important thing to understand is that Umbraco Models builder can not be used with items of type `IContent`, so do not expect any automated code generation when you working at this level within the CMS.

When you want to read or write data from `IContent` you will need to access all properties using generic `Get()` and `Set()` methods. Below gives a quick overview of how each operation works:

GetValue(string propertyTypeAlias) : This is the method on `IContent` that you will use the most frequently. Pass in a property alias and Umbraco will return the corresponding value:

```
var value = publishedContent.GetValue("myPropertyAlias");
```

GetValue()

When working with types of `IContent`, keep in mind that the underlining data structure will differ on an object by object basis. When your code is passed an item of type `IContent` it could be a media item, or a page created using a particular document type. The underlining construction type will determine its available properties.

Keeping this point in mind is important when you are writing `IContent` related code. If you try to access an invalid property in code, an invalid property exception can be thrown. To prevent your code from throwing exceptions, it is wise to do some property validation before you use `GetValue`() or `SetValue()`. This validation can be done with `HasProperty()`.

HasProperty(string propertyTypeAlias): As the name implies, `HasProperty()` allows you to check if a property exists before you perform an operation on it. Pass in a property alias as a `string` and get a `bool` back indicating if the property exists or not. This check does not include the state of the property, so it will return `true` even if the property has no actual assigned value.

SetValue(string propertyTypeAlias): It is also possible to perform write operations on items of type `IContent` using `SetValue(). To update an item, pass in the alias and the value:

```
publishedContent.SetValue("myPropertyAlias", "myValue");
```

SetValue()

Remember that you will still need to use the `Save()` method on `IContentService` before the change will be persisted in the database.

Asides from reading and writing data, the `IContent` type also exposes some other handy properties that are worth knowing about, these include:

ContentSchedule: Not all the pages within the CMS will be published. Likewise, not all published pages will stay live forever. For any Umbraco content, it is possible for a content editor to set a scheduled date and an expiry date. These scheduled action dates can be accessed using a property called `ContentSchedule`. This property will provide access to an item of type `ContentScheduleCollection`. Using an `enum` called `ContentScheduleAction` (which contains two options `Release` or `Expire`), it is possible to query this object to get or set these dates in code.

TemplateId: Gets or sets the template id that is associated to the item. Remember, a template is basically just a reference to an MVC view!

Published: Returns a `bool` that specifies if the item is published or not.

PublishedState: This property returns an `enum` whose value will either be `Published`, `Unpublished`, `Publishing`, or `Unpublishing`. I prefer using this property compared to the `Published` property, as I find it makes the code more declarative in nature.

Edited: Returns a `bool` that specifies if the content has been edited.

Blueprint: Returns a `bool` that specifies if the content item is a blueprint.

PublisherId: Returns the `Id` of the person who published the content item.

PublishDate: Returns a `DateTime` object of when the content was published.

PublishedCultures: Returns all the associated published cultures that have been created with the content item.

Umbraco.Extensions

On top of all the standard properties and methods exposed by `IContent`, Umbraco also provides some additional extension methods that can be used with items of type `IContent`. These extension methods can be found within a class called `ContentExtensions` within the `Umbraco.Extensions` namespace. Simply add `Umbraco.Extensions` as a using reference within the class where you are working with your item of type `IContent` and you will also get access to these methods:

GetWriterProfile() and GetCreatorProfile(): Gets the name of either the writer or the creator for the item. This method is more convenient compared to using `PublisherId` and then having to re-query the CMS

ContentStatus: This property is similar in nature to the `PublishedState` property. The main difference is that it exposes more information. This property will return an `enum` whose value can be either `Unpublished`, `Published`, `Expired`, `Trashed`, or `AwaitingRelease`.

TryGetMediaPath: Returns the media path for properties contained within the item of type `IMediaUrlGenerator`.

GetAncestorIds: Returns a collection of `int` that contains the `Id`s of all ancestors.

SetValue: The extension also exposes an added overload for `SetValue()`. This overload is specifically for media and file management. This new overload allows you to set the file value for any properties whose type is of file uploader.

Further Reading

- Content Vs Published Content
- How To Build A Search Page In Umbraco v9
- Umbraco Services Reference
- Information on the HttpContext class
- Information on NuCache
- Information on the UmbracoHelper
- Information on Examine

Hooking And Hacking Umbraco

The focus of the last few chapters has been on taking data from the CMS and rendering web pages. Not all development tasks on a web project will just be page related. Often you may bump into a requirement where you need to change the default Umbraco behavior when something happens. Common use cases where this need might arise include running some custom code when Umbraco starts, setting up a reoccurring task, configuring a custom service within the dependency injection framework and a whole heap of other coolness.

Umbraco is very easy to extend. Armed with a little knowledge, you can completely customize how the CMS behaves. To allow this, Umbraco provides many hooks that you can use in order to trigger whatever custom code that you would like to run.

The good news for Umbraco veterans is that most of the code to hook into the pipeline is predominantly the same from v8 to v9. Within Umbraco v8, a big change was made to the mechanics of how the CMS could be configured. Within v8, the concept of components and composers was introduced. This new architectural approach was designed with the future in mind. This is why not that much has changed between v8 and v9, despite the underlining framework upgrade.

The biggest change within Umbraco V9/10 that you will learn about in this chapter are notifications. Notifications were previously known as events within Umbraco V8 and below. Notifications are in essence a mechanism to hook into the Umbraco events pipeline.

When certain tasks are performed, either by a content editor within the CMS, or directly in code, in the background certain events (notifications) are also triggered. These notifications include page deleted, page saved, and page published events. Using notifications, it is possible for you to get whatever custom code that you need to run to trigger whenever one of these event types occurs.

Within this chapter, you will learn all about events and notifications and a whole deal more. You will learn about all the different ways that you can hook into the pipeline in order to hack Umbraco in any way you want.

The first step on this journey is looking at how you can register and trigger custom code to run within the pipeline. In order to register some custom code with Umbraco, you will need to learn about composers and components. This is where we will start on this chapter's journey. By mastering composers and components you can walk hand in hand along the beach of Umbraco mastery.

Composers and Components

Components and components will provide you with two different capabilities within Umbraco, however, both abilities are very closely related. A composer can be thought of as a registration class. By constructing one or more composers, you gain the ability to trigger stuff when Umbraco boots up.

As a standalone task, registration is a pretty pointless activity. In order to do anything meaningful, you also need the ability to register things. This is where components comes into play.

A component can not be called or accessed directly using a URL. Components can only be activated through registration within a composer. How you register a component with a composer will determine the order of when it is run. In most instances, you will register your components to trigger within the pipeline before any page controllers are called.

The terms components and composers stem from the composition paradigm. Within a compositional architecture, you have components (the parts), connectors (the devices that connect the parts together), and composition rules.

Within a Umbraco architecture, composers can be thought of as connector classes. The job of a composer is to provide the configuration rules that you require to connect a set of components together.

The official description of an Umbraco composer is something to compose functionality into the pipeline. Whenever you need to build some new functionality on the Umbraco start-up you will be creating a component and registering it within a composer.

If you read my Umbraco v8 book, I went into a lot of detail on how composers worked under the hood. That process has changed slightly in v9 compared to v8. In terms of development this change will not really impact you, however, it might be useful to know Within v8, there were different types of composers, this included `CoreInitialComposer`, `WebInitialComposer`, `ICoreComposer` and `IUserComposer`. These types have been deprecated within v9 and the only type that you now need to work with is `IComposer`. An example of how you can create a bare bones composer is shown below:

```
public class MyComposer: IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        // Register shizzle
    }
}
```

IComposer

To start registering your custom code with Umbraco, you will write your declaration code inside of `Compose()`. As part of the `Compose()` method signature, a single argument of type `IUmbracoBuilder` is defined. This builder object will allow you to start registering things with Umbraco. Out of the box, you can register lots of things including:

- Components
- Content Apps
- Notifications
- Dashboards
- Health Checks
- Hosted Services

Throughout the remainder of this book, we will constantly be coming back and touching on registering things within Umbraco. In later chapters, you will learn more about each one of these types, however, as the focus of this section is on composers and components the code we will cover here will all be component focused.

The good news is that regardless of the type of thing that you want to register, the code you need to write is very similar. To register something, pick the method related to the thing that you want to register, pass in the type you want to register as `T` and you are off to the races. The code to register a component would look like this:

```
public class RegisterComponentExample : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Components().Append<ExamineIndexManager>();
    }
}
```

Registering a component with a composer

In this example, as soon as this code is compiled and built into your project when the CMS starts up for the first time a component called `ExamineIndexManager` will be registered and triggered.

Asides from triggering some custom code, you may also need to factor in things like the order in which components are executed. It is also possible to further configure a composer using some special attributes, these attributes include:

`[ComposeBefore]` and `[ComposeAfter]`: When building a project within Umbraco you are free to create as many composers as you want. If you need to create a component that relies on state, or for a certain pre-task to have run, the ordering of execution is often key.

You can define the order in which your composers are run by decorating the class declaration with either the `[ComposeBefore]` or the `[ComposeAfter]` attributes. As you can imply from the names, `ComposeBefore` means the composer will run before something and `ComposeAfter` after. To define the dependency, simply pass in its type as `T`.

One word of caution when using these attributes is the possibilities of a cyclic exceptions. It is possible and pretty easy to accidentally create a circular dependency. A cyclic error can happen when composer one is configured to wait for composer two to run and composer two is configured to wait for composer one to run.

If you accidentally configured your composers this way, Umbraco will throw a StackOverflow error as it will get into an infinite loop. The code to define the order is shown below:

```
[ComposeBefore(typeof(ComposerOne))]
[ComposeAfter(typeof(ComposerOne))]
public class ComposerTwo : IUserComposer
{
    public void Compose(Composition composition)
    {
    }
}
```

Using ComposeBefore and ComposeAfter

AS you can see the code is simple enough. Decorate a class with the attribute and then use the type as the reference.

`[Enable]` and `[Disable]`: It is also possible to enable or disable a composer in code using the `[Enable]` and `[Disable]` attributes. As the name implies these attributes allow you either turn on or turn off a composer.

On their own, this ability is less useful for normal web developers. If you are a package developer, or, you want to override how some core functionality works, then the handiest use of these attributes is to disable a different composer whose functionality you want to override and change:

```
[Enable(typeof(ComposerOne))]
[Disable(typeof(ComposerOne))]
public class ComposerTwo : IUserComposer
{
    public void Compose(Composition composition)
    {
    }
}
```

Using Enabled and Disable

Components

Composers are used to register things, components are used to run custom code. You can create as many components as you need within Umbraco, however, my advice is to follow good software conventions and make sure each component you build has one clear purpose.

To build a component you will need to create a new class that implements the `IComponent` interface. `IComponent` defines two methods that you will need to implement, `Initialize()` and `Terminate()`.

`Initialize()` will be called on the components start-up. This is the method that you will be using most frequently. `Terminate()` is called on stop. In general, you can usually ignore the `Terminate()` and rely on the .NET garbage collector to do any clean up activities for you. Below shows an example of a component in action:

```
public class MyComponent : IComponent
{
    public void Initialize()
    {
    }

    public void Terminate()
    {
    }
}
```

IComponent

Within the `Initialize()` method you are free to add any new business logic you might need. If you are wondering what code you might want to add into a component, sit back and relax. As we continue our journey within this chapter and the rest of the book you will encounter lots of examples. The main takeaway should be an understanding of how you can register things and the mechanism of how to run custom code.

Dependency Injection

I think dependency injection is an essential task within any .NET build. In order to create an easy to maintain website, adhering to SOLID principles

and applying dependency injection ruthlessly anywhere you need to access custom code is key.

If you want to inject any of the custom classes and interfaces you create into a controller, or, service, you will need to understand how to configure those items within the Umbraco inversion of controller (IoC) container.

First, all dependency injection configuration can be done within a composer. Create a new composer and you will get access to the Umbraco `builder` object. From that object, you can then register dependencies. I typically create one composer within my projects called `RegisterDependencies` and add then add all my configuration code inside it.

How you configure this container has changed slightly within Umbraco V9/10. In Umbraco V7, there was no dependency injection support at all. This changed within v8, when Simple Injector was added to the core code. In v9, things have changed yet again. Within v9, Simple Injector has been replaced with the default .NET Core IoC container.

The good thing about Umbraco moving to a Microsoft standard is that it increases the chances that you will have used the dependency injection container before. This prevents us from needing to learn yet another framework just to use Umbraco.

When it comes to registering a custom class with the container, you have three main ways of configuring how the framework will inject the item when requested. These are:

Transient: When the framework injects a dependency using a transient connection, anytime a type is requested a brand new instance of that type will be created. When the framework encounters a constructor argument for a transient type, in the background it will instantiate a new version of the underlining class. The code to register a transient is simple enough:

```
public class RegisterDependencies : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Services.AddTransient<IDateService, DateService>();
    }
}
```

AddTransient()

If you want to follow good dependency injection practices you should use transient connections by default. The reason for this is that by creating a

new instance each time you will be less likely to hit any state management issues.

Scoped: In a scoped request a single instance of that type will be created and used for the duration of the current `HttpRequest`. The object will only be disposed of at the end of the current `HttpRequest`. Scoped requests will mean fewer objects are created during each request, however, the trade-off will be an object that is slightly longer-lived:

```
public class RegisterDependencies : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Services.AddScoped<ITutorialPageService, TutorialPageService>();
    }
}
```

AddScoped()

There is a higher risk with scoped requests around state. If the object's state is changed in one part of the system, when it is injected again into another area this previously altered state, may or may not be, what you expected.

Scoped requests can be slightly more performant, however, the performance gain is so marginal that I would not count it as a con. Basically, if you want to manage object state within a single request think about using a scoped request.

Singleton: Last up is the singleton mode. This mode is probably the most well understood. In this mode, a single instance of that type will be created for the entirety of the application's lifetime.

```
public class RegisterDependencies : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Services.AddSingleton<INavigationService, NavigationService>();
    }
}
```

AddSingleton()

Speaking from personal experience, singleton should be avoided as it can lead to extremely hard to track down bugs. Imagine you had a transient object that had two constructor arguments. Both of these arguments have been configured within the container as singleton requests. Even though the object is transient, it can still encounter state management is-

sues. In this small example this might seem like a very trivial example and easy to spot.

Imagine you inherit an existing Umbraco codebase that has hundreds of dependencies. Each dependency is configured separately. There is no way to look at the dependency configuration file and tell what's related and what's not.

The only way to debug these types of issues is to set a break-point, walk through the entry of your request and anytime a dependency is added check the types being passed it, then check their types. Repeat this process for anything that gets injected during the request. In a normal request, you might easily encounter over 100 items being injected. This is why spotting configuration mismatches are extremely hard and can lead to bugs.

If you have utility classes that never manage state, like a cache manager, Singletons can be useful. The advantage of singletons is better memory management, however, again these benefits are marginal nowadays as cloud computing is so powerful.

As you never get state management concerns when everything is set to a transient request, you can now see why transient is the recommended connection type to use!

Limits Of The .NET Container

If you have previously used other dependency frameworks, like StructureMap, you will likely already know and love their auto discovery and auto registration capabilities. These two things combined will prevent you from having to write lots of boring boilerplate registration codes within your container. Not only keeping you sane but also saving you time.

Auto-discovery and auto registration combined will give the container the ability to automatically try to find and register any custom code that you have created that you want to register with the framework. Configuring the container to automatically register things for you means that you can avoid having to write registration code each and every time you create a new class.

Sadly the default .NET Core container does not provide this capability. The .NET container was built to be simple, so, if you want to have this sort of easy dependency management capabilities within your Umbraco project, you will need to install an additional component.

There are a few free packages that add this capability to the default .NET Core container. The one that I have used and can validate works well is called Scrutor. Scrutor can be configured in lots of ways, however, for this section, I will just cover the code that will work for most people out of the box. If you find that this configuration does not work for you, I recommend reading the Scrutor documentation as you will likely need to tweak this config a little:

```
public class RegisterDependencies : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Services.Scan(scan => scan
                .FromAssemblies(
                typeof(IAppSettingsService).Assembly,
                typeof(IWebsiteUrlsService).Assembly,
                typeof(IHomepageService).Assembly
                )
                .AddClasses()
                .UsingRegistrationStrategy(RegistrationStrategy.Skip)
                .AsImplementedInterfaces()
                .WithTransientLifetime());
    }
}
```

Configuring Scrutor()

After installing Scrutator, create a composer and add the above config. The first step towards getting automatic dependency registration to work is that you need to give the tool a list of assemblies that you would like the tool to scan. This list will usually include your website assembly, your core library assembly (if you have one) and anything else where you have code you want to automap.

You can register an assembly within Scrutor using `FromAssemblies()`. The easiest way to register an assembly is to pass in its type. Knowing how to reference your custom assemblies in code can be tricky, so to make life easy, there is a shortcut.

The shortcut is to use reflection to get access to the type. Passing a type as `T` into `typeof(T)` and using the `Assembly` property will give you access in code to the containing assembly type. You can use this code to easily get a reference to any containing assembly for any object type. You can pick any interface or class to use as `T` within the containing assembly that you want to include.

In the example code above, the type `IHomepageService` lives in the website project, `IWebsiteUrlsService` in the core project, and `IAppSettingsService` within an additional library called Common. All it takes is these three lines to include all of the projects custom class libraries to be scanned

With all of your assemblies registered, you need to tell the framework to try to map your custom dependencies, this is done using `AddClasses()`. While registering things, if Scrutor encounters the same interface twice it will throw an exception. By including `.UsingRegistrationStrategy(RegistrationStrategy.Skip)`, Scrutor will not throw an exception.

To create an auto-mapping from class name to interface, e.g. `MyClass` to `IMyClass`, you use `AsImplementedInterfaces`. Finally to register everything as transient requests, use `.WithTransientLifetime()`. These 6 lines will save you from writing a lot of code, so I recommend you implement it!

Tweak the assembly registration part to fit your solution and everything should magically work. No need to manually register anything else ever again!

Notifications

Understanding how to register things is the first step towards modifying Umbraco's default behavior, the second step is to knowing how to hook into an event. Whenever a content editor logs into the Umbraco backend and performs certain actions, in the background events are triggered. Traditionally within Umbraco, these actions have been called events, however, as previously mentioned, these have now been re-branded as notifications!

By hooking into a notification, you can trigger some custom code of your choice to run whenever Umbraco triggers that action. This type of feature can be really handy when you need to introduce some new capability to Umbraco that is not provided out-of-the-box.

Umbraco exposes over 40 different types of notifications that you can hook into. You can see a complete list of all of these notifications from this page within the Umbraco Github. Below lists some of the groups that I have used the most:

- Content Saved
- Content Published
- Content Deleted
- Media Saved
- Media moved
- User password changed
- Login failed
- Emptying Recycle BinNotification

Registering A Notification Handler

In order to hook into a notification, you will need to create a notification handler and register it with a composer. You can register your notification handler within an existing composer, or if it makes you happy within a new one, the choice is yours.

The steps that you will need to follow in order to create a composer that will register a notification handler are exactly the same as previously listed. Create a new class that implements the `IComposer` interface, implement the `void Compose(IUmbracoBuilder builder)` method and

then using the `builder` object, access the `AddNotificationHandler` handler method:

```
public class RegisterNotifications : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        // Register Notification Handler
        builder.AddNotificationHandler<ContentPublishedNotification, SubscribeToContentPublishedNotifacations>();
    }
}
```

How to Register a Notification

The `AddNotificationHandler` signature uses generics which expects two different types of `T` to be supplied. The first `T` is the notification type that you want to hook into. The second is the custom notification handler type that you want to register with the event.

The skeleton code required to build a custom notification handler is very easy: Create a new C# class and make it implement from `INotificationHandler`:

```
public class SubscribeToContentSavingNotifacations : INotificationHandler<ContentSavingNotification>
{
    public void Handle(ContentSavingNotification notification)
    {
    }
}
```

Creating A New Notification

Make sure that the name of the class maps to the type that you pass into the second argument within `AddNotificationHandler`. The notification type needs to marry up between the composer and the notification handler, otherwise, your notification handler will not dun, d'oh!

With the skeleton code set-up, you are free to add your custom code into the `void Handle(T notification)` method. `Handle(T)` defines one incoming argument which will be the argument type that you specified as the first `T` as part of the notification registration code within the composer.

The underlining type will vary based on the handler that you want to hook into. In general, each different notification will provide you access to the item(s) that triggered the event. For example, the `ContentSavedNotification` event exposes a property called `SavedEntities` and the `DeletedNotification` event exposes a property called `DeletedEntities`. Each property will return a collection of data that is related to the event.

The property name that you will need to access will vary depending on which notification handler you are using. As each notification type only exposes one property, I am confident you will find it obvious which property you need to target.

```
public void Handle(ContentSavingNotification notification)
{
    foreach (var node in notification.SavedEntities)
    {
        // Do something
    }
}
```

Accessing the content that triggered a notification

Each property will give you access to the related data as type `IContent`. As the item is of type `IContent`, you can easily read and update the item as needed. You could update content, media and member data, using methods like `GetValue()` and `SetValue()`.

Reoccurring Tasks

When working on a website build, you may often bump into a requirement where a task needs to be repeatedly executed over a period of time. Common use cases where you might bump into this type of requirement could include syncing logs, pulling data from a third party, nudging a cache, or cleaning up a database table before it gets too large.

Within Umbraco, it is possible to define a reoccurring task in code using a Umbraco task runner and a reoccurring task object. To create your own reoccurring task, the first thing you need to do is create a class that inherits from an abstract base class.

For the old-timers be aware that the name of this base class has changed in V9 compared to v8. Previously you would have created a class that inherited from an abstract base class called `RecurringTaskBase`. Instead in V9/10, the class that you now need to inherit from is called RecurringHostedServiceBase.

The constructor signature for `RecurringHostedServiceBase` is also a little different. You will need to supply two `TimeSpan` parameters to the base constructor a period and a delay. The period defines how often the task should recur, while the delay defines the interval for the task to the first time after Umbraco starts.

The code below will create the task runner. The purpose of this runner is pretty pointless. The task will simply write a message to the log file every 5 minutes:

```
public class MyLogger : RecurringHostedServiceBase
{
    private static TimeSpan Delay => TimeSpan.FromMinutes(10);
    private static TimeSpan Period => TimeSpan.FromMinutes(10);

    private readonly ILogger _logger;

    public MyLogger(ILogger<MyLogger> logger)
        : base(Period, Delay)
    {
        _logger = logger;
    }

    public override Task PerformExecuteAsync(object state)
    {
        _logger.LogInformation("Logged Message");
        return Task.CompletedTask;
    }
}
```

How to Setup a reoccurring task

Registering a task runner with Umbraco is done by a composer. The code to register a task within a composer looks like this:

```
public class CleanUpYourRoomComposer : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Services.AddHostedService<MyLogger>();
    }
}
```

How to Setup a background task register runner

Thats all there is to creating a scheduled tasks in Umbraco.

Content Finders

When it comes to website construction, the more tools that you add to your toolbox the better equipped you are. Learning about content finders will make you a more well rounded Umbraco developer, however, content finders will be one of your least used tools. I would go as far as to say that it is highly likely that you may never need to create or use a content finder in anger throughout your Umbraco career!

In the first few chapters of this book, you learnt how routing works within Umbraco. The fundamental aim of routing is mapping a known URL to a controller and an action. In order for this mapping to work, Umbraco uses slightly different routing rules compared to vanilla MVC. An incoming request needs to be mapped to a page within the CMS before a controller can be triggered. As part of this process, Umbraco needs to look up pages within the CMS and this is where content finders come into play.

To map an incoming URL to an Umbraco item within the CMS a look-up is performed. As part of this look-up process, Umbraco iterates through a collection of content finders. As soon as a successful content match is made within a finder, the request can be deemed successful.

For a successful match scenario, the data held by Umbraco about the item can be added to the request context and eventually, a corresponding controller can be called. If none of the finders are able to successfully match the request, a 404 error will be thrown.

A content finder and a routing rule both have distinct purposes, however, they clearly can be considered within the same ballpark in terms of purpose within the pipeline. The purpose of a routing rule is to map a URL to controller. The purpose of a content finder is to map the Umbraco item within the CMS before that controller is called.

Out-of-the-box, Umbraco contains a number of default content finders including `ContentFinderByUrlAlias`, `ContentFinderByIdPath`, and `ContentFinderByUrl`. These finders are all run whenever an incoming request is made. Each content finder needs to implement a method called `TryFindContent()`. It is within `TryFindContent()` where the look-up code is added.

As a developer, it is possible to create your own custom content finder. If you decide to go down this path, create a class that implements from `IContentFinder` and then add whatever look-up logic that you see fit within `TryFindContent()`. Obviously, do not forget to register your finder with a composer!

The only time that I have personally needed to use a content finder in the real-world was when building a website that had a pretty non-standard requirement. This company was essential promoting content for a bunch of teams. The architecture required that each team had an area within the website. The page structure for each team was exactly the same and

the only real differentiator was a few bits of data that came from an external API.

Instead of forcing the content editors to keep the pages within the CMS and the data contained within the API in synch, I created a custom content finder to perform a mapping based on the data held within this external API. If the team name existed in the API, a page would be rendered. The content finder allowed content editors to manage content at the API layer which saved them a lot of time.

The situation where I needed to use a content finder was pretty niche. If you need to manage content from an API, or, you have a complex multi-language need, a content finder may come in handy. I would not hold my breath that you will need to use one very often! The code to create a content finder is shown below:

```
public class MyContentFinder : IContentFinder
{
    public bool TryFindContent(IPublishedRequestBuilder contentRequest)
    {

        var requestUri = contentRequest.Uri;

        // query the cache for match, if found set the request
        contentRequest.SetPublishedContent(matchedContent);
        return true;
    }
}
```

Creating a content finder

Within `TryFindContent()` add any custom look-up logic you need. If you can not find a match return `false`. If you can find a match return `true` and then either use `SetRedirect()`, `SetResponseStatus()`, `SetPublishedContent()`, `SetTemplate()`, or a combination to change the request.

Within Umbraco 10 there was a slight change to this methods signature. Within 10, content finders were updated so that they could work asynchronously. To create a content finder in v10, use this revised code instead:

```
public class PageNotFoundContentFinder : IContentFinder
{
    public Task<bool> TryFindContent(IPublishedRequestBuilder request)
    {
        return Task.FromResult(true);
    }
}
```

Creating a content finder in v10

If you want to use `SetPublishedContent()` you will need to supply it within an item of type `IPublishedContent`. Going back to my example, if you want to render data from an external API, you will need to create some dummy pages within the CMS to make this work.

Finally, registering a new finder with Umbraco is done using a composer:

```
public class RegisterContentFinder : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.AddContentFinder<MyFinder>()
    }
}
```

Registering a content finder

Job done!

Errors - 404 and 500s

The last type of route hacking that will be covered in this chapter is error handling. Despite great developers and testing processes, things do go wrong in production. When requests do not go as planned, presenting the user with friendly messaging is a much better user experience compared to showing them the default .NET error pages.

It can be very easy to forget to configure graceful error handling on a project, so remember to add it within your project plan. You will need to en-

sure that your site can gracefully handle two types of errors, a page not found request (404), and a full-blown temper tantrum (500).

Some websites go to great lengths to create an interesting and creative not found error page. The good thing with Umbraco is that you can do the same easily as 404 pages can be created inside of the CMS so content editors can use the full power of the Umbraco property feature set to get those creative juices flowing.

Technically, there is nothing special about creating a 404 page inside of the CMS. The process of creating a 404 page is exactly the same as any other page. You can use an existing document type, like a content page or create a 404 specific document type, the choice is yours.

Remember, a document type alias can not start with a number. This means that you can not name a document-type `404`. Most implementors who decide to create a dedicated 404 document type call it something like `PageNotFound` or something equally as creative.

After creating the document type, you will then need to generate the model's builder model and create all the corresponding scaffolding, e.g view model, controller and view. With the standard page creation tasks covered, create the 404 page within the content tree, publish it and test that it works.

The next step is to configure Umbraco so that the pipeline loads this page whenever a 404 error is thrown. The simplest way to configure the 404 page is from within `appsettings.json`. You will need to add some configuration within the `Umbraco` node:

```
"Umbraco": {
    "CMS": {
        "Error404Collection": [
          {
            "Culture": "default",
            "ContentId": "1071"
          }
        ]
      }
    }
}
```

ContentId

There are a few handy things to note about this JSON. First, `Error404Collection` can handle an array of options, meaning you can define multiple 404 pages. This can be a very handy feature on multi-language websites. It is possible to return a specific language-aware 404 page by varying the culture key. You should always supply a `default` option though!

The 404 page above is configured based on page Id, however, this is not the only option. I recommend you favour using the page Id as you will find it easier to locate the 404 within the CMS. You can add the Id to the internal URL to access it. It is also possible to configure the 404 either by the page GUID or an XPath expression:

ContentId

Using XPath in previous versions of the CMS was really handy because dealing with scenarios like multiple start pages per culture was not as easy to configure as it is now. From v9 onwards, I'm not sure there's a good use-case for using XPath, especially as there is also a more code-oriented solution for solving more complex 404 routing logic.

There is also another special type of content finder worthing knowing about called `IContentLastChanceFinder`. Setting up and configuring an `IContentLastChanceFinder` is exactly the same as the steps within the last section. Create a new class and implement from `IContentLastChanceFinder`.

Within the `TryFindContent()` method, add the 404 look-up logic that makes you happy. Using C# to do that complex logic will be much easier than trying to mess around with XPath queries. The code to create an `IContentLastChanceFinder` is shown below (remember that the content finder signature is slightly different in v9 compared to v10):

```
public class PageNotFoundContentFinder : IContentLastChanceFinder
{
    public Task<bool> TryFindContent(IPublishedRequestBuilder request)
    {
        var notFoundPage = // Code to get the 404 page within Umbraco.js
        request.SetIs404();
        request.SetPublishedContent(notFoundPage);

        return Task.FromResult(true);
    }
}
```

IContentLastChanceFinder

Like everything else, you would need to register this content finder within a composer:

```
public class SetContentLastChanceFinderComposer : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.SetContentLastChanceFinder<PageNotFoundContentFinder>();
    }
}
```

SetContentLastChanceFinder

The next time someone attempted to view an invalid URL the item you returned from within this method would be triggered. I am doubtful that you will ever need to use this technique, however, it might be a useful tool to know about one day.

500 Error Pages

The next type of error handling will cover situations where things have gone really wrong and an exception has been thrown. The status codes of these types of errors will usually be 500. When a 500 error has been thrown this is usually an indication there is a code issue somewhere. Like the 404 page, it is possible to use a Umbraco page to deal with 500 errors, however, I do not advise this. A good practice is to always return the 500 error as a static HTML page.

Using a code generated page to handle code errors can result in infinite loops. Imagine you made a mistake in the menu which throws a 500 error. Umbraco seeing the error tries to load the 500 page. While the server is

trying to create the 500 page, it tries to load the menu. This request throws another 500 error, which results in another 500 page lookup. When this happens a Stackoverflow exception will be thrown. If too many Stackoverflow exceptions get thrown within a certain time span, the whole web server will turn off. This happens to protect the whole underlining server from falling over.

Creating a static 500 error page is simple enough. Within `Start` add this line within `Configure()`:

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseExceptionHandler("/500.html");
    }
}
```

UseExceptionHandler

Add a corresponding HTML page within your`wwwRoot` and your 500 scenarios will now be solved!

Further Reading

- Composing
- Components
- Dependency injection in ASP.NET Core
- Dependency Injection Deep Dive Within Umbraco V9
- Umbraco Notifications explained

Customise The Backend

The Umbraco backend provides a ton of features for content editors, however, sometimes you may need to provide some functionality that is not covered out the box. Within this chapter, we will look at all the different options that are available for you to extend the backend.

Umbraco is a very customizable CMS. As a framework, you can add custom screens, dashboards, sections, trees and content apps in order to extend the CMS with any new functionality of your choosing.

One interesting nuance with Umbraco backend development is that you have two potential development paths. When it comes to building custom views and screens, Umbraco supports creation using AngularJs and C#. If you are an AngularJs master, it will be useful for you to be aware of this alternative route.

This books focus will only be on backend development using C#. Personally, I do a lot of frontend development using React and Javascript. I do not know a lot of AngularJs and I do not have the appetite to learn a whole new framework just to do the odd Umbraco task once in a blue moon. The C# way of building things will be applicable to every Umbraco developer and it is the most well-worn path.

If you really want to, you can find a lot of information online about using AngularJs and Umbraco. Using AngularJs can make more sense if you are a package developer and you have less control over when the code is compiled. When you have the power to compile the code, I think sticking to C# will give you a more consistent codebase that anyone on your team can work with.

Sections

The first step in adding new functionality within the Umbraco backend is to start adding your own custom links within the CMS to trigger custom screens. The first link that we will add will be in the most prominent area,

After logging into the backend, at the top of the page, you will see a ribbon that typically contains 8 sections. The default sections will include a content tab, a media tab, a settings tab, a packages tab, a users tab, a members tab, a forms tab and a translations tab. The first customisation within this chapter will allow you to add your own custom tab within this ribbon.

Within Umbraco terminology, these tabs are known as sections. Creating a custom section on its own is not that useful. Defining a custom section will add a new entry within the ribbon with a link, however, the definition will not include any implementation details about what should happen when someone clicks on that link.

The takeaway is that there are two development tasks when building a section. The first is to create a class that will register the custom section with Umbraco. As a separate task, you will also need to build something else in order to render a custom screen. This could be a vanilla MVC page, a dashboard, or a tree view. The options and the implementation details around rendering custom screens will be covered shortly.

As a developer, there are a few ways that you can define a custom section. I will cover the C# path, however, be aware this is not the only path. The alternative option involves creating a folder within `App_Data` and creating a `package.manifest` file. A `package.manifest` file is a custom Umbraco file. The `package.manifest` file is constructed out of JSON and is an alternative way of defining a section or a package. You can find more information on how to do this at the Umbraco website.

To create a section, create a new class that implements from the ISection` interface:

```
public class MyCustomSection : ISection
{
    public string Alias => "myCustomSection";

    public string Name => "My Custom Section";
}
```

Creating a section in code

`ISection` defines two properties. The `alias` will be the way you can reference the section in code. The `name` property is self-explanatory. This custom class will need to be registered with Umbraco before the section will display. This registration is done using a composer:

```
public class RegisterUmbracoComponents : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        builder.Sections().Insert<MyCustomSection>(1);
    }
}
```

Registering a section with a composer

One useful thing to note with this code is the integer that is being passed into the constructor. The section registration method defines an optional sort order parameter. This argument can be used to position the section wherever you want within the ribbon.

When it comes to rendering anything custom within the backend, you will bump into a resource file issue. If you just used the code above and tried viewing the backend, your section name will look a little funky. You will notice that the name property is not used. Instead, the sections `alias` will be rendered wrapped in brackets. This happens because you also need to add the display value within a resource file.

You can add resource files within `App_Plugins`. Within `App_Plugins`, create a new folder and call it anything you like. I tend to call mine `CustomResourceLabels`.

Next, create a new sub-folder called `lang`. Finally within `lang` , create a new `xml` file. The name of this XML file needs to map to the culture you have set within the CMS. For reference, the default culture is `en-US` so you would name the file `en-US`.xml`. In this file, you need to add this config:

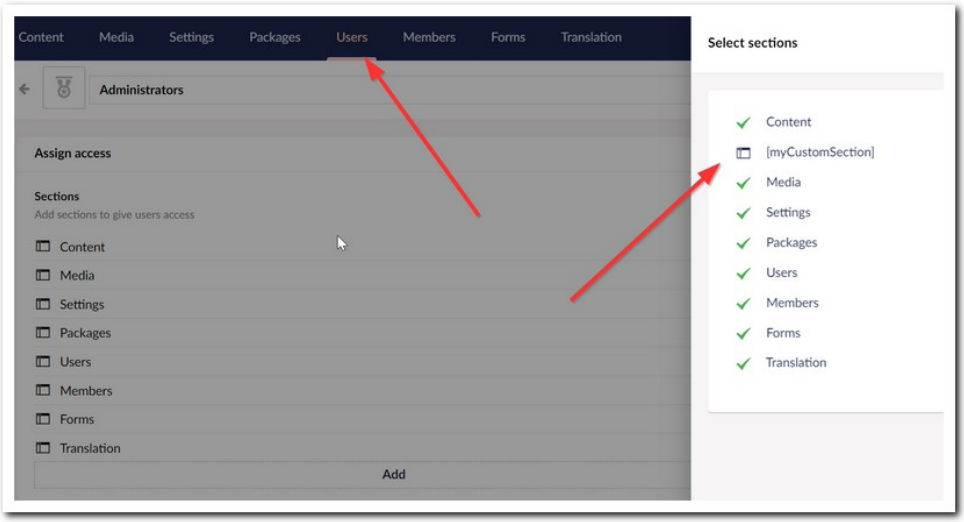
```
{
    "sections": [
        {
            "alias": "myCustomSection",
            "name": "Snazzy Title"
        }
    ]
}
```

Setting label text

Within the XML, you need to map your section `alias` to the value that you want to appear within the backend. In this example, the text `Unpublished Tutorials` will be rendered whenever a call to an alias called `UnpublishedTutorials` is made.

There is also another big gotcha when it comes to sections. Just because the section is registered with Umbraco that does not mean that it will appear within the CMS. Before a content editor can view it, you will need to add the section to a permission group. The permission groups related to a particular content editor will determine which items are rendered within the top ribbon.

To add a section to a permission group, within the backend open the `Users` section. Click on the `Groups` icon in the top left. By default, you should be presented with a list of 5 groups. Click on the group that you want to allow access to the section, for example, `Administrators`:



Assigning a section to a permission group

Within the `Assign access` area, click on the `Add` button. Locate your section and add it to the list. Click `Save`.

Assuming you have logged in as an administrator when you refresh the page in your browser the section will appear.

If you click on the section, you will see a very dull grey page. To spice this page up, you will need to create an additional component. This might be a custom tree or dashboard.

Dashboards

The easiest way to extend the Umbraco backend with a custom view is to create a custom dashboard. A custom dashboard will allow you to render a custom screen within the backend. By being able to build a custom screen, you can provide extra capabilities to your content editors that will allow them to do whatever you want. This could include writing some code to connect to a third-party SQL database which saves and retrieves custom information.

To create a new dashboard you will need to create a new class that implements from the IDashboard interface. Unlike other components, you do not need to register your dashboard with a composer, however, you can access a `Dashboards` property from within a composer if you need to do anything advanced.

`IDashboard` comes with five properties that you will need to implement:

- **Sections**: This property defines wherein the backend you want the dashboard to appear. Note, that you are also free to include more than one area!

To access any default Umbraco aliases, you can use a handy helper method called `Constants`. The `Constants` class is found within the

`Umbraco.Cms.Core` namespace. `Constants` exposes a property called `Applications`. `Applications` exposes the aliases for the content, packages, media, member, settings, translation, users, and forms sections.

If you have created a custom Umbraco section as described in the last section, you can also add its alias within this list. This is how we change that dull grey screen into something more useful!

I will use the `Content` section in this example.

- **AccessRules**: This property is used to define which content editors are able to view the dashboard. You can get access to the different access rights from within `Constants.Security` (also found in `Umbraco.Cms.Core`). In this example I want the dashboard to be viewable to all content editors, so I use `Constants.Security.EditorGroupAlias`
- **Alias**: Like all things Umbraco, your dashboard will need a unique `alias`
- **View**: The view property defines the Url that Umbraco will use to access the screen

Building The Controller, Model, View

We have defined a dashboard that when clicked will call a URL. The next step is to build a custom screen that triggers whenever a request to that Url is made. To do this we can use C# to create a controller and a view. The important thing is to ensure your controller has the correct permissions so only content editors can access it. You don't want any old Tom, Dick or Harry accessing your secure dashboard.

The official way to lock a controller for backend use only, is to use the `UmbracoAuthorizedController`. I had trouble getting this working. When I tried to return a `View` type from this controller I encountered this error:

```
The name 'View' does not exist within the current contex
```

Instead, I created my own controller and defined the permissions manually. This is easy enough. If you look at the source code (here), you will see the `UmbracoAuthorizedController` is just a normal controller with two attributes. We can clone its functionality like this:

```
[Authorize(Policy = AuthorizationPolicies.BackOfficeAccess)]
[DisableBrowserCache]
public class MyBackendController : Microsoft.AspNetCore.Mvc.Controller
{
}
```

Custom BackOffice Access Controller

Within this controller, you can use normal C# and MVC in order to do whatever functionality you want and to render whatever view makes you happy.

Admin Layout: When you render your view within the backend, for a consistent UI experience, you will want it to look like every other Umbraco backend page.

This is why you should create a new master layout and add the Umbraco CSS and JS calls that an out-of-the-box backend Umbraco view uses. The HTML to do this is shown below. Within this new layout, within the HTML head section you need to reference the correct CSS files:

```
<head>
    <meta charset="utf-8">
    <meta name="viewport" content=
"width=device-width, height=device-height, initial-scale=1, maximum-scale=1, user-scalable=no">
    <meta charset="UTF-8" />
    <meta name="robots" content="nofollow" />
    <link href="~/rs-plugin/css/settings.css" media="screen" rel="stylesheet">
    <!-- CSS -->
    <link href="~/css/bootstrap.min.css" rel="stylesheet">
    <link href="~/css/style.css" rel="stylesheet">
    <link href="~/css/admin.min.css" rel="stylesheet">
    <link href="~/css/font-awesome.min.css" rel="stylesheet">
    <link href="~/fontello/css/fontello.css" rel="stylesheet">
</head>
```

Reference correct CSS

Within the body tag of the layout, you need to wrap the render call with this HTML:

```
<div class="container-fluid">
    @RenderBody()
</div>
```

Use correct HTML structure

Finally, you need to make sure you are calling the correct scripts at the bottom of the page:

```
<div class="container-fluid">
    @RenderBody()
</div>
```

Reference correct Javascript filers

For this example, I have added all this HTML inside of a layout view called `AdminLayout.cshtml` that was created within the root `View` folder. Any subsequent custom screens that you create, you can simply reference this layout from the corresponding view in order to get the correct Umbraco UI styling for your custom screen:

Creating AdminLayout file

Routing Rules

The important thing when setting all this up is our good friend routing. Remember, routing is different in a vanilla .NET project compared to a CMS project. To get the routing to work in a secure Umbraco request, there are two important things you need to ensure when setting this up:

- The Url to access the screen needs to start with `umbraco/backoffice/plugins/`
- You need to create a rule within the routing table otherwise you will get a 404 error

You can add rules directly within `start.cs` although I do not recommend this. To ensure a clean codebase, I recommend you add your rules within an extension class and then reference the extension within `start.cs`. The code to create this extension is shown below:

```
public static partial class UmbracoApplicationBuilderExtensions
{
    public static IUmbracoEndpointBuilderContext UseCustomRoutingRules(this
IUmbracoEndpointBuilderContext app)
    {
        app.EndpointRouteBuilder.MapControllerRoute(
                        "AdminDefault",
                        "umbraco/backoffice/plugins/<CONTROLLER>/<ACTION>",
                        new { Controller = "<CONTROLLER>", Action = "<ACTION>" });

        return app;
    }
}
```

Umbraco Application BuilderExtensions Example

Within `Start.cs` you access the extension ike this:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseUmbraco()
    .WithEndpoints(u =>
    {
        u.UseCustomRoutingRules();
    });
}
```

Use Custom Routing Rules Example

Remember, the route Url needs to start with `umbraco/backoffice/plugins/` otherwise it will not work!

When creating a dashboard, you will also encounter the same resource file issue. The same issue that occurs when creating a custom section. You will need to add a corresponding resource file entry for the dashboard alias before it will render correctly.

The process of registering a resource file is the same. If you have already created a folder to deal with resources use that one. Adding a new file is done within `App_Plugins`. Create a folder within `App_Plugins` named whatever. Create a `lang` folder. Create a new `xml` file whose name is based on the default culture. In this file, you need to add this config:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<language>
  <area alias="dashboardTabs">
    <key alias="UnpublishedTutorials">Unpublished Tutorials</key>
  </area>
</language>
```

Setting label text

The important line is the one that maps the dashboard `alias` to a value that you want to appear within the backend. In this example, the text `Unpublished Tutorials` will be rendered whenever a dashboard with an alias of `UnpublishedTutorials` is made.

Trees

The next custom backend component that you can create is called a tree. You will have used a core Umbraco tree object pretty much every time you have logged into the Umbraco backend. Tree items are basically any node that lives within the left side bar in the backend. An example of a core Umbraco tree object, is the document type area within the settings section.

A tree is associated against a section (core or custom). After a tree is associated to a section, when viewed, the tree will render in the left-hand sidebar. A tree is defined in code by creating a class and inheriting from `TreeController`.

When defining a tree, you determine what nodes will be rendered inside of it. You may choose to render a single item, or a collection, the decision is yours. Each node that you add to the tree is completely customization. This customization includes children and actions.

Defining a node is cool, however, you also have the power to decide what actions a content editor can perform against each node. Whenever you hover your mouse over a node within the Umbraco backend, three ellipses will appear next to it. Clicking on the ellipses will bring up a context menu. The good news is that you have full control on a per node basis to configure what menu items will appear and what custom views will be triggered when that menu item is clicked.

I have some bad news, it is not possible to load a Razor/C# view after a node action has been triggered. Tree views have to be created using HTML and AngularJs. Only being able to create tree action views with AngularJs is a pain. Having to learn a whole framework just to create a tree seems like a high-level of effort with a minimum gain.

If you live in the camp that is unfamiliar with AngularJs, I recommend you favour using dashboards. Within your custom dashboard screen you can use a CSS framework like Bootstrap or Tailwind to build a custom tree yourself. I personally tend to use Umbraco dashboards instead of trees what I need to render a custom screen in the backend. I find dashboards a little quicker to build and easier for the development team to manage. The code to create a tree is shown below:

```
[Tree(
    "myCustomSection",
    "myCustomTree",
    TreeTitle = "My Custom Tree",
    TreeGroup = "group",
    SortOrder = 5)]
[PluginController("myCustomSection")]
public class MyCUstomTreeController : TreeController
{

    private readonly IMenuItemCollectionFactory _menuItemCollectionFactory;

    public MyCUstomTreeController(
        ILocalizedTextService localizedTextService,
        UmbracoApiControllerTypeCollection umbracoApiControllerTypeCollection,
        IMenuItemCollectionFactory menuItemCollectionFactory,
        IEventAggregator eventAggregator)
        : base(localizedTextService, umbracoApiControllerTypeCollection, eventAggregator)
    {
        _menuItemCollectionFactory = menuItemCollectionFactory;
    }

    protected override ActionResult<TreeNodeCollection> GetTreeNodes(
        string id,
        [ModelBinder(typeof(HttpQueryStringModelBinder))] FormCollection queryStrings)
    {
        var nodes = new TreeNodeCollection();
        if (id == Constants.System.Root.ToInvariantString())
        {
            var node = CreateTreeNode(
                                "myId",
                                "-1",
                                queryStrings,
                                "Option One",
                                "icon-list",
                                false);

            nodes.Add(node);
        }

        return nodes;
    }

    protected override ActionResult<MenuItemCollection> GetMenuForNode(
        string id,
        [ModelBinder(typeof(HttpQueryStringModelBinder))] FormCollection queryStrings)
    {
        var menu = _menuItemCollectionFactory.Create();
        return menu;
    }
}
```

Creating A Tree Controller

The `GetTreeNodes()` method is where you can add the code that will define what items will be rendered within the tree. You can define one or more nodes as the `GetTreeNodes()` works with collections. For each

item that you add to the tree, you can define an id, the parent Id, the node title, the icon to render in the tree, if the item has children and a custom route.

By default, when a content editor tries to view a node within a tree Umbraco an edit action event is triggered. When an event action is triggered, the framework tries rot load a default view called `edit.html`. You will need to create this view within your webroot at this location otherwise nothing will display:

`App_Plugins` -> `myCustomSection` -> `backoffice` -> `myCustomTree` -> `edit.html`

Where `myCustomSection` maps to the value you put within the `PluginController` attribute and `myCustomTree` maps to the tree alias you are creating.

The edit action is not the only action that you can define. It is also possible to associate different actions with a node. Defining what actions are available is done within a second method exposed within the base `TreeController` class called `GetMenuForNode()`.

Within `GetMenuForNode()` you can add any actions you want in code. Out-of-the-box, Umbraco provides about 30 different action types that you can use. These out-of-the-box actions include:

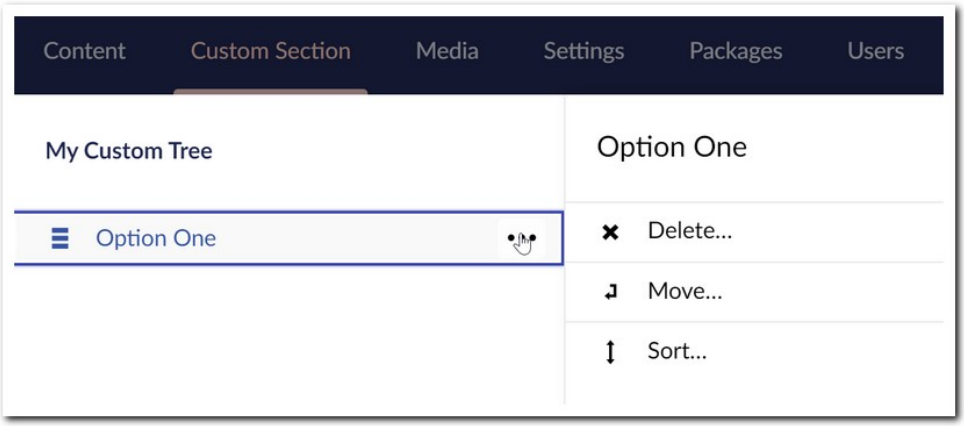
- Copy
- Delete
- Publish
- Sort

The complete list of supported actions can be found here. An example of how to add a delete, move and sort action to a node in code is shown below:

```
protected override ActionResult<MenuItemCollection> GetMenuForNode(
    string id,
    [ModelBinder(typeof(HttpQueryStringModelBinder))] FormCollection queryStrings)
{
    var menu = _menuItemCollectionFactory.Create();
    menu.Items.Add<ActionDelete>(LocalizedTextService, true, opensDialog: true);
    menu.Items.Add<ActionMove>(LocalizedTextService, true, opensDialog: true);
    menu.Items.Add<ActionSort>(LocalizedTextService, true, opensDialog: true);
    return menu;
}
```

Adding Actions

This code would then translate to these options within the CMS:



Tree Action Nodes

When any of these actions are triggered from within the CMS, a corresponding view will be called within the plugins folder. The `delete` action will trigger a request to load the `delete.html` file. The good thing about using a tree node is that you can hook into these different notifications.

What you do within each view will be custom to your needs. The view will need to be AngularJs based.

Content Apps

The final way of rendering a custom screen within the Umbraco backend is via a Content App. A content app is a custom tab that you can make appear within the page editing view within the backend. You are free to create as many content apps as you need. Whenever a content editor creates a new page, or, edits an existing page within the CMS, for each content app you define a corresponding tab will appear next to the 'Info' tab.

After an editor clicks on a content app, its corresponding factory will be called. Within this code, you can then add whatever logic that you want with the end goal of triggering a custom screen. It is then left up to your creativity to decide what to create. The options are pretty endless, you could create an app that displays page related analytics, links to social sites, or whatever your tiny little mind can invent.

To create a content app, the first step is to create a new class and inherit from `IContentAppFactory`. After doing this, you will need to implement its `GetContentAppFor()` method. Within `GetContentAppFor()`, you will need to add the code to trigger the view and pass any contextual data into it. Finally, to register a new component with Umbraco, you will need to register the app using a composer.

To follow a good architectural approach, it's recommended to make the content factory render request go via a controller rather than adding all corresponding logic within the view. Doing this makes for a cleaner codebase, however, it adds an additional step into the mix.

Umbraco does not provide any specific content app-based controllers. This means that you will need to use a vanilla MVC controller with a custom routing rule to make the factory talk to a controller. Whenever you use a normal MVC controller within Umbraco you will not get any contextual information about the current page object.

As we want to solely use C# to create this content app, we have a dilemma. Without passing the associated page data into the controller, the content app will not have any context about the page around it. There is no `CurrentPage` object, or, any way to access information about the current page within the controller. At a minimum, we will need to access the current page Id within your content app controller. How do we get this?

You do not need to get context about the current page from inside of the controller. Instead, you can pass the page Id from `IContentAppFactory` to the controller. This is possible due to how `IContentAppFactory` is called.

Whenever a content editor tries to edit a page in the CMS, on each request the custom `IContentAppFactory` class will be triggered. When a render request occurs, the current page's context will be passed into the custom `IContentAppFactory`. You can then access this data and pass it onto the controller. An example of the code to create a factory with the page Id being passed to the controller is shown below:

```
public class AdditionalLinksContentApp : IContentAppFactory
{
    IHostingEnvironment _hostingEnvironment;
    UriUtility _uriUtility;

    public AdditionalLinksContentApp(IHostingEnvironment hostingEnvironment)
    {
        _hostingEnvironment = hostingEnvironment;
        _uriUtility = new UriUtility(_hostingEnvironment);
    }

    public ContentApp GetContentAppFor(
        object source,
        IEnumerable<IReadOnlyUserGroup> userGroups)
    {
        var content = source as IContent;
        if (content != null)
        {
            return new ContentApp
            {
                Alias = "additionalLinksApp",
                Name = "Links",
                Icon = "icon-cloud",
                View = _uriUtility.ToAbsolute($"/umbraco/backoffice/plugins/contentapp/links/{content.Id}"),
                Weight = -100,
                Active = true
            };
        }

        return null;
    }
}
```

IContentAppFactory

You can see this magic on Line 24, where I pass the page `Id` into the Url. The key thing towards getting this working is routing. In order to get the routing to work, you will need to add a custom routing rule into your ap-

plication's route table. With the rule in place, the controller will be called and it will be passed the page `Id` as an action argument. Using this `Id` you can then get information about the page using the normal Umbraco APIs.

As we are working with a backend screen, you will also need to consider security. You will not want any Tom, Dick, or Harry accessing your private data. When creating a content app controller, you will also need to add the correct authorisation checks to it. To implement a secure backend screen within Umbraco, the Url to trigger the content app controller needs to start with `umbraco/backoffice/plugins/`. The code to register an appropriate routing rule is shown below:

```
public static partial class UmbracoApplicationBuilderExtensions
{
    public static IUmbracoEndpointBuilderContext UseCustomRoutingRules(this IUmbracoEndpointBuilderContext app)
    {
        app.EndpointRouteBuilder.MapControllerRoute(
            "ContentApp",
            "umbraco/backoffice/plugins/contentapp/links/{id}",
            new
            {
                controller = "AdditionalLinks",
                action = "Index",
                id = UrlParameter.Optional
            });

        return app;
    }
}
```

UseCustomRoutingRules

This rule then needs to be registered within `Start.cs` like this:

```
public class Startup
{
    public void Configure(IApplicationBuilder app,IWebHostEnvironment env)
    {
        app.UseUmbraco()
            .WithEndpoints(u =>
            {
                u.UseCustomRoutingRules();
            });
    }
}
```

Startup

Another point worth mentioning is that Umbraco will also not load your custom `IContentAppFactory` class magically, you will need to register it using a composer:

```
public class RegisterNotifications : IComposer
{
    public void Compose(IUmbracoBuilder builder)
    {
        // Content Apps
        builder.ContentApps().Append<AdditionalLinksContentApp>();
    }
}
```

IComposer

With the content app registered with Umbraco and the correct routing rule enabled. The rest of the code is pure C# MVC!

Creating An Umbraco Content App

The first step is to create a controller:

```
public class MyContentAppController : Microsoft.AspNetCore.Mvc.Controller
{
    public Microsoft.AspNetCore.Mvc.ViewResult Index(int id)
    {
        var viewModel = new ViewModel();
        return View(viewModel);
    }
}
```

Content App Controller Example

Implement an `Index` action and add your custom code. With the code nailed, it's time to create an associated view. Create a new view here:

`Views` -> `MyContentApp` -> `Index.cshtml`

Within the view add whatever custom logic you need. With that done your content app should now load!

Further Reading

- More information on Sections
- Creating a custom Dashboard

- More information on tree actions
- Create a kick-ass ContentApp in Umbraco 9
- Create a custom backend Dashboard in Umbraco V9

Members and Editors

Unlike some of the newer, best-of-breed CMS systems on the market, Umbraco provides more than just content creation capabilities. In terms of a complete CMS offering, Umbraco goes several steps further. One of these steps is the ability to build a members-only area completely inside of the CMS.

When I'm working with a client who is looking to change their current CMS, Umbraco's built-in membership capabilities can often be the main deciding factor in why they pick Umbraco from the alternatives.

In this new world of best-of-breed, many of the competitors will make you purchase an additional service like Auth0 in order to handle member management. Auth0 is great, however, for clients who want to keep the budget down, this extra complexity and extra cost can rule them out as a contender.

The good news is that the membership offering provided by Umbraco is strong. Out of the box, Umbraco comes with all the capabilities, APIs and components that will allow you to manage members within the CMS. You have complete control over what data you want to store against a member profile. You also have complete control over how those members are managed whether it be on an individual level or on a group level.

Umbraco also provides all the authorization and authentication capabilities that you will need. The benefit of this is that on the front end, it is relatively straightforward to create features like a log-in component, or a restricted area.

Within this chapter, you will learn how to configure, set up, manage and build a members area within your solution. The first step when building a members area is defining the member profile. What information will you need to store about a member in order to target and market to them correctly?

The nice thing about working with members within Umbraco is that you will already be familiar with most of the concepts required to master this topic. Members are defined in exactly the same way as content is created.

After you create members within Umbraco, a copy of that person's data will be added to NuCache. As member data is also stored within NuCache, the way in which you access that data is the same as when you want to access content, via the Umbraco context. The steps that you need to follow in order to access the context in code are exactly the same. After accessing member data from the cache, you will also be working with another familiar concept, objects of type `IPublishedContent`.

As you can hopefully see, the concepts that we learnt at the start of the book will now start to reap their rewards.

Members And Roles

Within Umbraco, we have the concept of editors and members. The difference between the two types of accounts is relationships. Editing accounts should be given to your staff and internally trusted users. An editing account will have the permissions to access the Umbraco backend. Editors will include your developers, marketing team and content creators.

Umbraco also provides the capability for you to manage external users of your website. These types of users, known as members, will typically be your customers. Every member that you create inside of the CMS will be a known person to you, as they will need some form of identifier, like a username or email address.

Typically, when you want to implement member functionality on a build, it is to provide something like a My Account area, or maybe even a secure members portal. The members feature can also be used for implementing personalisation.

Members and editors are accessed differently in code and within the backend. Members are managed within the CMS from the `Members` section, while editors are managed within the `Users` section. To access data about a member you would use the `IMemberManager` service. To access information about an editor you would use `IUserService`.

When it comes to Umbraco development, the chances are a lot higher than you will need to work with members compared to editors. Out of the box, Umbraco uses a default member schema called `Member`. The `Member` schema defines all the custom properties that can be associated with an individual member. The default profile contains properties like is approved, failed password attempts, password question and password answer. To view the `Member` schema, go to the `Settings` section and open the `Member Types` node. Under the `Member Types` node, you should see an entry called `Member`.

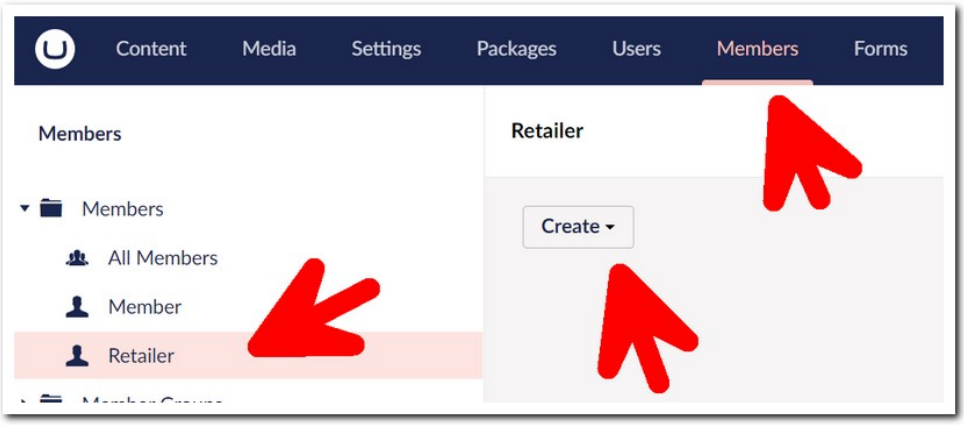
If you need to add additional properties to the default member schema this is not a problem. It is possible to either extend the default profile or, create a new one and use that instead. If you find yourself in this situation, I recommend that you create a new profile rather than modify the default profile.

If you try to create a profile from scratch, you will likely need to copy properties from the default profile which simply wastes your time. The safest option would be to clone the default schema and then add any custom properties to your new profile. Cloning means that you do not accidentally mess up the default profile and you also inherit all the required properties that Umbraco needs to work correctly!

The steps to perform member modelling onto a member profile are the same as the steps for content modelling with a document type. You can create a member profile and give it an alias. From the member profile builder, you can associate different tabs and properties to that schema.

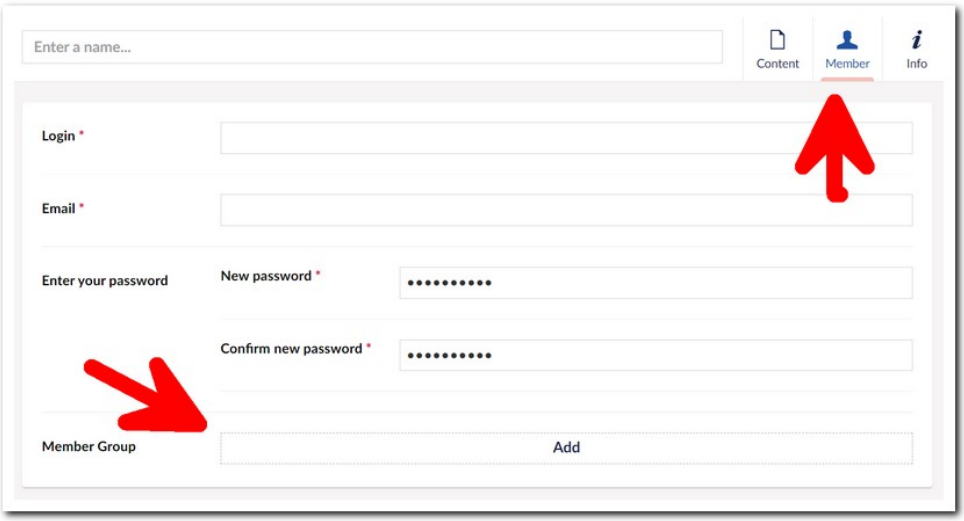
The data types and properties that you can add to a member profile are the same ones that were covered in the content modelling chapter. This is all possible because the member profile, like a document type, is created as `IContent`.

In essence, everything that you can do on a document type, you can do on a member profile. The only real difference that you might notice when editing a member profile is that you will not have access to any publishing, or UI related properties. After you create and save your new member profile, you can use the profile to create new members. To do that, go to the `Members` section and within the `Members` node, click on your new profile and click on the 'Create' button and you will be taken to the create new member page.



Create New Member

From this page, you will notice a tab called `Member`. From this tab you can add the `Login`, `Email` and `Password` for the member. At the bottom of this screen you will also see an option called `Member Group`.



Member Group

When building a members area, you will likely need to deal with permissions. On many membership sites, not all members are created equally.

You may want to allow certain members to have more access than others.

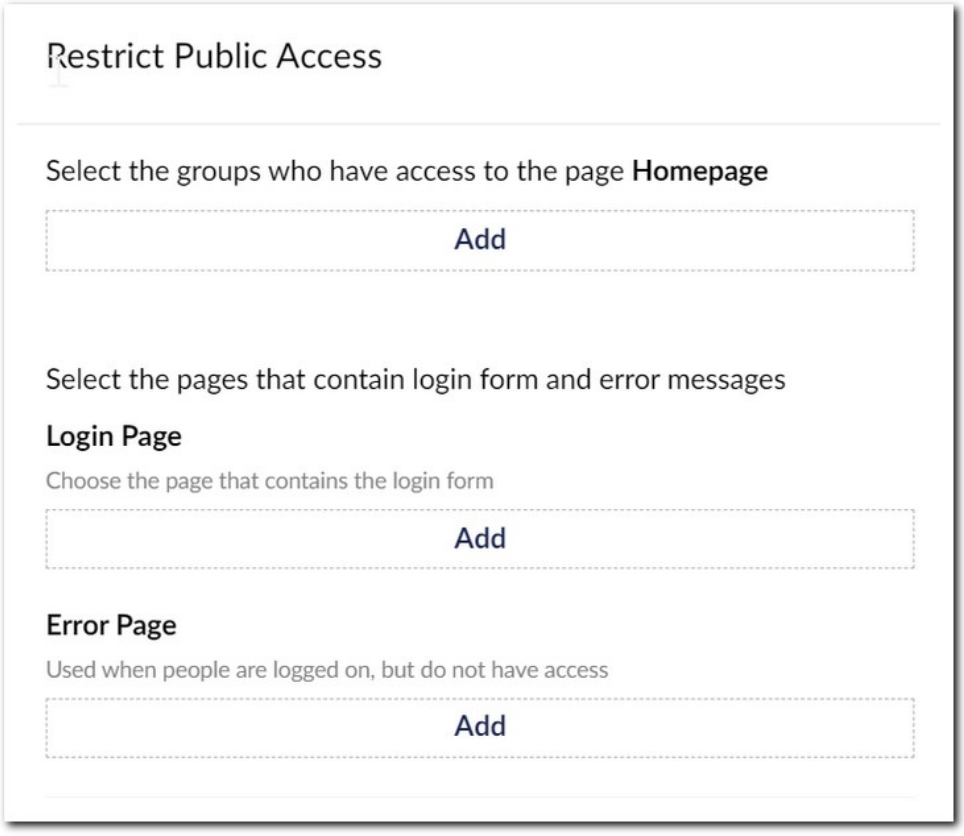
When dealing with permissions, I recommend that you work on a role basis rather than trying to assign permissions on a per-member basis. The `Member Group` property on the member's tab is the property that will allow you to assign which roles that particular member is associated with.

The process of creating a new role is very easy. Within the `Members` section, right-click on the `Member Groups` node so the open context menu appears. Click on the `Create` option. From the create new member group screen, give the role a name, click `Save` and job done.

After defining members and associating them to appropriate roles,the next step is to create the secure members area. By default, all pages within the CMS are created as public. Anyone with that pages URL can access it. To restrict a page, you restrict public access to it.

You can apply page access restrictions from the site tree within the `Content` section. Locate the page that you want to prevent from having public access, then right-click on that page, so the context menu appears. At the bottom of the context menu, you should see an option called 'Restrict Public Access'.

When restricting a page you have the option to restrict it either by specific member protection, or by group based protection. Always use group-based protection as it will save you massive amounts of content management hassle.



Member Group

When you restrict a page by role, by default, all public access to that page will be disabled. Only members who are logged in and are also assigned to the corresponding role(s) will be able to access that page.

While setting up these permissions, you will also need to define a login page and an error page. Umbraco will use the login page to automatically redirect site visitors who try to access the page and are not logged in. Umbraco will use the error page when logged in members try to access the page, but, they are not included within an allowed role.

Login Component

Whenever an unauthenticated access request is made to a page where the public access role has been removed, Umbraco will automatically redirect that request. The page that has been configured within the login page property will be the page Umbraco attempts to load. As Umbraco does not ship with its own login page, it is left up to you to build out that page.

When it comes to building a login page, you have two options. You can either build a dedicated document type that completely handles the login flow or, you could build a component that can be added to an existing document type.

As we are dealing with security, I recommend that you always opt to build a dedicated login document-type. When security is concerned, you do not want to give a content editor the power to accidentally break things. Your security mantra should always be towards making the system as restricted as possible.

The benefit of using a dedicated document type is that it will be much easier to control how it is used. By restricting which properties are editable and by writing additional validation checks, like a notification handler, you can ensure that the login process will always work as you intend it to. Also as an aside, never add caching to this document-type!

Within this example, I will show you how to create a login document type that can deal with two states. For anonymous requests, a login box will be rendered. For previously authenticated requests, some private content will be displayed instead.

This approach is obviously not the only way to architect how a login flow behaves, however, the example will give you a solid foundation to build a flow to meet most requirements. As the process is left up to you, simply update the code to match your requirements.

I recommend that you build your login page with four main components.

Login document type: The first thing that you will need to build is a document type to handle the login request. Just like creating any document type, you will also need to create a corresponding controller, view model and view. The controller can be built using `RenderController` which

means you do not need to register anything custom within the routeing table.

In terms of content modelling, I find the login document type usually ends up being fairly light as its main purpose is to be a skeleton for the login flow. Most of the content gets created within sub-components.

When building your login page, I recommend that you also consider creating and registering two related notification handlers. One that prevents the page from accidentally being deleted. This check can be made by hooking into ContentDeletingNotification.

The other handler should prevent two instances of that type from being created. After all, why do you ever need two login pages inside of the CMS? This can be achieved by hooking into `ContentSavingNotification` and using `IContentService`.

Create a document type called `LoginPage` or something equally as imaginative. Refer to the creating document type chapter if you are unsure how to do this. After creating the document type and generating the corresponding model builder model, the next step is to create an associated controller:

```
public class LoginPageController : RenderController
{
    private IMemberManager _memberManager;
    private IPublishedValueFallback _publishedValueFallback;

    public LoginPageController(
            ILogger<LoginPageController> logger,
            ICompositeViewEngine compositeViewEngine,
            IUmbracoContextAccessor umbracoContextAccessor,
            IPublishedValueFallback publishedValueFallback,
            IMemberManager memberManager
        )
        : base(logger, compositeViewEngine, umbracoContextAccessor)
    {
        _publishedValueFallback = publishedValueFallback;
        _memberManager = memberManager;
    }

    public override IActionResult Index()
    {
    }
}
```

Login Page Controller

The code to drive the membership feature looks like this:

```
public override IActionResult Index()
{
    var loginPage = new LoginPage(CurrentPage, _publishedValueFallback);
    var viewModel = new ComposedPageViewModel<LoginPage, LoginPageViewModel>
    {
        Page = loginPage,
        ViewModel = new LoginPageViewModel
        {
            IsUserLoggedIn = _memberManager.IsLoggedIn(),
        }
    };
    return View("~/Views/LoginPage/index.cshtml", viewModel);
}
```

Login Controller Action

In terms of membership-related code, the interesting part of this code is the injection of `IMemberManager`. `IMemberManager` is the Umbraco API that will allow you to do all sorts of cool member related things.

You can use the `IsLoggedIn` property to get the members' logged-in status. Based on this state, you can then decide which view component to load within the view. Create a view using Umbracos standard naming conventions:

`Views` > `LoginPage` > `Index.cshtml`

In this view, add this HTML:

```
@model ComposedPageViewModel<LoginPage, LoginPageViewModel>
@{
    Layout = "../master.cshtml";
}

<section id="main">
    <div class="container">
        <div id="content">
            @if (Model.ViewModel.IsUserLoggedIn)
            {
                @(await Component.InvokeAsync("AuthenicatedComponent"))
            }
            else
            {
                @(await Component.InvokeAsync("AnonymousComponent"))
            }
        </div>
    </div>
</section>
```

Login Page Razor HTML

Depending on the member's status will determine what content will be displayed on the page. If a user is not logged in, render the `AnonymousComponent` and the login form should be displayed. If the member is authenticated, render the view component that spits out the restricted content. The HTML for the login form will be created inside `AnonymousComponent`:

Anonymous view component: Instead of building the login form within the document type, I find it better to build the form within a view component for a nicer separation of concerns.

View components are the way to render components within ASP.NET 5. View components replaced the partial views of the ASP.NET Framework. You do not need to add any custom routing rules to create a view component. You simply need to create a controller and an associated view. This is the HTML I added to the view to render the login component:

```
@model Umbraco.Cms.Web.Common.Models.LoginModel

@using (Html.BeginUmbracoForm<JonDJones.Core.Controller.SurfaceControllers.LoginBoxController>("Authenticate", FormMethod.Post))
{
    @Html.AntiForgeryToken()

    <div class="col-12">
        @Html.ValidationSummary()
    </div>
    <div class="col-12">
        @Html.LabelFor(m => m.Username)
        @Html.TextBoxFor(m => m.Username, new { placeholder = "Username" })
    </div>
    <div class="col-12">
        @Html.LabelFor(m => m.Password)
        @Html.PasswordFor(m => m.Password, new { placeholder = "Password" })
    </div>
    <div class="col-12" style="margin-top: 1rem">

        <button name="login" class="form-button-submit button icon solid fa-envelope" type="submit">
            Login
        </button>
    </div>
}
```

Begin Umbraco Form

It is possible to design your website so the member area section is rendered on an additional page. I'm doing everything on the same page in this tutorial to make it easier to follow rather than best practice. You could just as easily do a redirect to another page in the `SurfaceController` (we will get to that code shortly). The corresponding view controller component is pretty simple and it doesn't really need to do anything clever:

```
public class AnonymousComponent : ViewComponent
{
    public async Task<IViewComponentResult> InvokeAsync()
    {
        return View(new LoginModel());
    }
}
```

Login View Component

Authenticated view component: The sole purpose of this component is to render content for people already authenticated. If you do not need any content within this component to be content editable, you do not have to hook it into the CMS. The code to create the `AuthenticatedComponent` view controller is shown below:

```
public class AuthenicatedComponent : ViewComponent
{
    private IMemberManager _memberManager;

    public AuthenicatedComponent(
            IMemberManager memberManager
        )
    {
        _memberManager = memberManager;
    }
    public async Task<IViewComponentResult> InvokeAsync()
    {
        var profileData = await _memberManager.GetCurrentMemberAsync();
        return View(new ProfileViewModel
            {
                Name = profileData.Name,
                Email = profileData.Email,
                Comments = profileData.Comments,
                Roles = profileData.Roles,
            }
        );
    }
}
```

Authenticated View Component

This is the view:

```
@model JonDJones.Core.ViewModel.Poco.ProfileViewModel
<article>
    <a href="@Url.SurfaceAction("Logout", "LoginBox")" class="button icon solid fa-file">
        Logout
    </a>
</article>
```

Authenticated View

A surface controller: The job of this controller is to deal with the login form data being posted back to the server. The surface controller is the connective tissue of how the form posts data back to Umbraco, how you validate login submissions and what to do with the request. The code to create an authentication controller is shown below:

```
public class LoginBoxController : SurfaceController
{
  IMemberSignInManager _memberSignInManager;
  IMemberManager _memberManager;

  public LoginBoxController(
      AppCaches appCaches,
      ServiceContext services,
       IProfilingLogger profilingLogger,
      IPublishedUrlProvider publishedUrlProvider,
      IUmbracoContextAccessor umbracoContextAccessor,
      IUmbracoDatabaseFactory databaseFactory,
      IMemberManager memberManager,
      IMemberSignInManager memberSignInManager)
          : base(umbracoContextAccessor, databaseFactory, services, appCaches, profilingLogger, publishedUrlProvider)
  {
      _memberManager = memberManager;
      _memberSignInManager = memberSignInManager;
  }

  [Microsoft.AspNetCore.Mvc.HttpPost]
  [Microsoft.AspNetCore.Mvc.ValidateAntiForgeryToken]
  public async Task<IActionResult> Authenticate(LoginModel model)
    return CurrentUmbracoPage();
  }
}
```

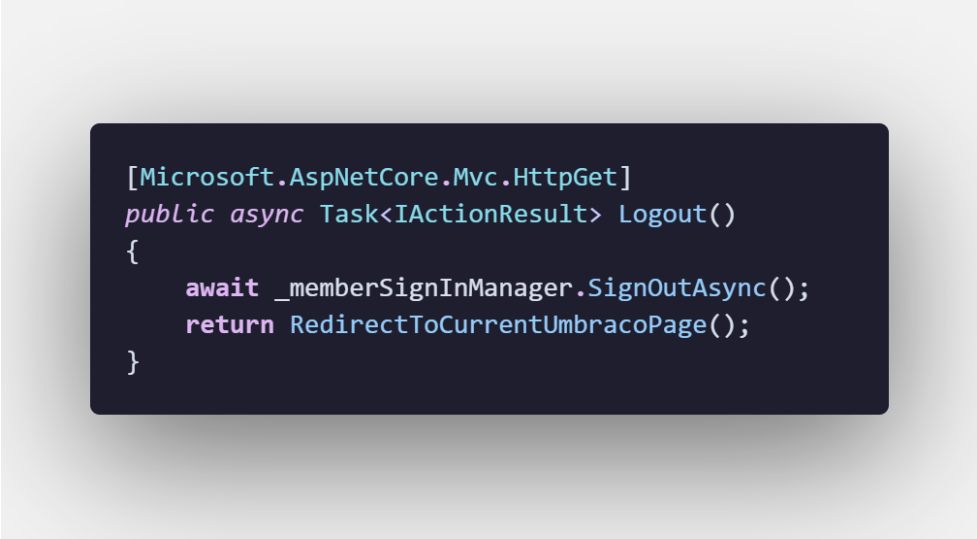
Surface Controller

To check if a site visitor has entered a valid credential, you can use a combination of `IMemberManager` and `IMemberSignInManager`. You use the `ValidateCredentialsAsync()` method within `IMemberManager` to check if the passed in details that have been supplied are valid. If the credentials are valid, you can use the `PasswordSignInAsync()` method within `IMemberSignInManager` to log that person into Umbraco:

```
[Microsoft.AspNetCore.Mvc.HttpPost]
[Microsoft.AspNetCore.Mvc.ValidateAntiForgeryToken]
public async Task<IActionResult> Authenticate(LoginModel model)
{
       if (!ModelState.IsValid)
    {
        ModelState.AddModelError(string.Empty, "Entry Denied Pal!");
        return CurrentUmbracoPage();
    }

    var isValid = await _memberManager.ValidateCredentialsAsync(model.Username, model.Password);
    if (isValid)
    {
        var isSignedResult = await _memberSignInManager.PasswordSignInAsync(
            model.Username,
            model.Password,
            true,
            false);

        if (isSignedResult.Succeeded)
        {
            return Redirect("members-area.html");

        } else if (isSignedResult.IsLockedOut)
        {
            // do something
        }

    }

    return CurrentUmbracoPage();
}
```

Surface Controller Authenticate

When creating the surface controller, do not forget to decorate the authentication action with the `ValidateAntiForgeryToken` attribute. This will make sure naughty people can not do naughty things with your form. To make this work you need to make sure you have the `@Html.AntiForgeryToken()` herlp within your forms opening and closing tage@



```
[Microsoft.AspNetCore.Mvc.HttpGet]
public async Task<IActionResult> Logout()
{
    await _memberSignInManager.SignOutAsync();
    return RedirectToCurrentUmbracoPage();
}
```

Surface Controller Logout

It might look like it takes a lot of code to get it up and running, however, when you start coding it out you'll find that it is not too much. Conceptually I think that as long as you understand you need a page, two components and the `SurfaceController` the rest is fairly easy. When I first tried building a site I thought the `SurfaceController` should be the page controller. Within Umbraco, the `SurfaceController` is the thing that deals with data postback only.

Managing Members In Code

In this section, we will delve a little deeper into managing members at the code level. In terms of member capabilities not much has changed within Umbraco v9, the same can not be said under the hood!

Umbraco now makes use of ASP.NET Core Identity instead of ASP.NET Membership. Like almost every other .NET Core component, Identity is configured within `Startup.cs`. Identity supports all the usual capabilities like user, role, password, and profile data management. Identity can also be configured to support an external login provider like Google.

The takeaway is that if you need to extend how your editors, or members log into Umbraco the steps have changed compared to previous versions. Identity is considered an upgrade over ASP.NET Membership because you can do more with it. Covering every aspect of Identity is outside the scope of this book. As Identity is a generic Microsoft feature rather than a Um-

braco specific component, you can head over to MSDN to learn more about it.

The handy thing within V9/10 is that Umbraco completely takes care of this switch for you. You can simply add members as normal, use `IMemberManager` and live in blissful ignorance of the underlining changes.

When it comes to working with members in code, you can opt to work on the username/password level, or, you can opt to work with `MemberIdentityUser`. `MemberIdentityUser` is an Umbraco specific extension of the .NET core `IdentityUser` type. The `MemberIdentityUser` object exposes all the same features and properties as `IdentityUser` as well as some more Umbraco specific properties like, `LastLockoutDateUtc`, `MemberTypeAlias`, `Name`, `Id`, and `IsApproved`. You can tell by the `IMemberManager` method signatures where and when you can use each type.

To build the login form in the last section, we needed to make use of `IMemberManager` and `IMemberSignInManager`. Both of these services are new within v9. In v8, to get similar functionality you used to use `MembershipHelper`, however, this has now been deprecated.

There is a difference in purpose between `IMemberSignInManager` and `IMemberManager`. Use `IMemberSignInManager` when you need to log members in or out. `IMemberSignInManager` provides three methods `PasswordSignInAsync()`, `SignInAsync()`, and `SignOutAsync()`. The code to log someone out using `IMemberSignInManager` is shown below:

```
public class LoggedIn
{
    public LoggedIn(IMemberSignInManager member)
    {
        member.SignOutAsync();
    }
}
```

Code for Sign out

The other API, `IMemberManager` should be used when you need to access, edit and manage member data on an individual level. `IMemberManager` does not provide you with the functionality to log people in or out, instead, it exposes features like

`GetCurrentMemberAsync()`, `IsLoggedIn()`, `FindByIdAsync()`, `FindByEmailAsync()`, `FindByNameAsync()`, and `AsPublishedMember()`.

GDPR and consent

Nowadays, whenever you are creating a member area, you need to consider consent. The General Data Protection Regulation (GDPR) legislation passed in 2018 legally demands it!

In essence, GDPR means legally you need to follow certain guidelines when holding data about members. The law is comprised of 7 principles, lawfulness, transparency, purpose, accuracy, storage, integrity and accountability.

This book is written by a developer, for developers. I'm definitely not a GDPR lawyer and all my advice within the section is not legal advice. All I can say is that you will definitely need to consider GDPR when designing your membership area and if you are unsure of what you need to do, ask a lawyer!

For us as implementors, GDPR often means implementing some form of consent management capability. You will need to ensure members can opt in and opt out of things. Within v9 and onwards, the Umbraco team makes this type of management much easier with the inclusion of the `IConsentService` API.

The `IConsentService` API will allow you to register, revoke and view member consent. The most common use-case to deal with consent is around marketing preferences. Are you allowed to send someone an email? Within Umbraco, providing this type of feature is easy enough with `IConsentService`:

```
public MemberService(IConsentService consentService)
{
    var user = "jon";
    var consentAction = "marketingOptIt";

    var newConsent = consentService.RegisterConsent(
        user,
        "Our.Custom.Umbraco.Plugin",
        consentAction,
        ConsentState.Granted);
}
```

IConsentService.RegisterConsent()

To register a consent, you use `RegisterConsent()`. Pass in the username, a context, and an action. After registering a consent, you can check a users consent status using `LookupConsent()`:

```
public MemberService(IConsentService consentService)
{
    var user = "jon";
    var consentAction = "marketingOptIt";

    var consents = consentService.LookupConsent(user);
    consents?.Any(x => x.Action == consentAction && x.State == ConsentState.Granted);
}
```

IConsentService.LookupConsent()

Combining `IMemberSignInManager`, `IConsentService`, and `IMemberManager` will allow you to build a pretty powerful and feature-rich membership site!

Creating, Editing and deleting members

The Umbraco APIs that have been covered so far will allow you to manage users at an individual level, however, these APIs will not allow you to create new members, member profiles, or, roles. To do that you can make use of IMemberService and IMemberGroupService. Both APIs can be found within the `Umbraco.Cms.Core.Services` namespace, the difference between the two is if you want to perform actions on a member or role level.

IMemberService is to edit and create members. `IMemberService` exposes methods like `CreateMember()`, `DeleteMembersOfType()`, `GetAll()`, `GetMembersByMemberType()`. An example of how to create a member in code using this API is shown below:

```
public class MemberService
{
    public MemberService(IMemberService memberService)
    {
        var member = memberService.CreateMember(
            "username",
            "test@test.com",
            "jon",
            "memberAlias");
    }
}
```

IMemberService

`IMemberGroupService` provides the ability to create, edit and delete roles in code. `IMemberGroupService` exposes methods like `Delete()`, `GetAll()`, `GetByName()`, and `Save()`.

```
public class MemberGroupsExample
{
    public MemberGroupsExample(IMemberGroupService groupService)
    {
        var groups = groupService.GetAll();
    }
}
```

IMemberGroupService

Combining `IMemberService` with `IMemberGroupService` will allow you to build your own bespoke member management screens.

User Service

The focus of this chapter has been on members, however, it is also possible to manage content editors at a code-level as well. When you want to work with content editors/users you will need to use the IUserService. `IUserService` can be found within the `Umbraco.Core.Service` namespace and in terms of capabilities it is comparable to `IMemberManger`.

The user service provides all sorts of methods to manage content editor accounts in code. You can do pretty much every admin task possible using `IUserService`, including resetting passwords, creating new content editor accounts, updating profiles and even disabling and removing accounts completely from the system!

These operations are possible due to the methods it exposes, like `ValidateLoginSession()`, `ClearLoginSessions()`, `Delete()`, `GetProfileByUserName()`, `GetUserById()`, `GetPermissions()`, `GetPermissionsForPath()`, `AssignUserGroupPermission()`, `GetAllInGroup()`, `GetUserGroupByAlias()`, `Save()`, and `DeleteUserGroup()`.

Like most of the APIs within Umbraco 9, to access the API within code, you can access it using dependency injection:

```
public class LoggedIn
{
    public LoggedIn(IUserService userService)
    {
        var _userService = userService;
    }
}
```

Code for Sign out

In the real world, it is less likely that you will be re-creating screens that provide the same features that Umbraco ships with out of the box. I have never needed to use the `IUserService` in anger in a project yet, however, if you do have a need, this is the API you should start with.

Further Reading

- How to Create a Members Area Within Umbraco V9
- IMemberManager
- Information on the member service
- Information on ASP.NET Core Identity
- Information on Notifications
- What is GDPR, the EU's new data protection law?
- Marketing and consent

Multi-Language Websites

I assume that you do not keep up-to-date with the latest e-commerce research, so I will share a stat with you. A recent study has proven that around 75% of global consumers prefer to buy products in their native language. This means that if your client transacts in a global market and wants their website to convert optimally with their entire customer base, they will need to write multi-lingual content.

Historically, Umbraco was not great in terms of the out-of-the-box multi-language capabilities that it provided. Previously, you had to install and rely on third-party community packages in order to provide multi-language support, however, from Umbraco 8 this changed.

Umbraco 9 provides everything that you may need when building a multi-language website. You do not need to worry about having to rely on any unofficial third-party packages anymore. You are free to build wherever you like and have confidence that when it comes to upgrading, you will not need to completely rewrite your site.

When it comes to deciding on how you want to build your multi-language website, there are a few different architectural routes that you can take. For me, picking the most appropriate route to take, depends on the content.

On a multi-language project, the first step to success is getting a clear answer on the company's long-term content strategy. The content strategy will determine how you will need to build components like the primary menu and the language selector. The content strategy will also determine how you should organize and structure the top-level items within the content tree. Finally, the strategy will also determine what global settings may (or may not) need to be enabled.

The important aspect of the content strategy to ascertain is if each language variation will need to support a completely different content structure, or not. If the content structure is vastly different between languages, you will need to treat each language variation as a new website.

To apply this pattern within the CMS, you will need to define a top-level homepage for each language that you need to support. Content editors can then add whatever pages in whatever order they want under these homepages, without impacting any other language variation.

Structuring the content tree like this will allow editors to organically grow the content for each language, without encountering any limitations. When you need to support independent content growth, this is the best architectural route to follow.

The only main downside for content editors when applying this pattern is the potential duplication of effort. If there is a requirement to mirror big sections of pages between the sites, the editor will need to create all of these pages under each appropriate homepage manually. Imagine your

site supported 10 language variations. Most of the page structure is different, however, 30 pages are similar. Doing some simple maths, that's 300 pages the editor would need to create and maintain inside of the CMS!

The only other way to structure the content tree for multi-language is to define a single homepage within the CMS and then associate different language content on a per-property basis. In terms of content maintenance for editors, this route has a lot less set-up. Create a page once, add content, change the language, then rinse and repeat until all content has been added, job done.

The limitation when picking this approach is that the content structure will be identical forever more. If the company decides that they want to move to a more organic content structure, the whole content tree will need to be rewritten!

In terms of development effort, this second route is easier and simpler. At a code level, the only main differentiator is the culture code being requested. Following this path will be very similar to building a normal single language site, just with a few added extras. The umbraco APIs have been built to support this type of page creation.

In terms of development effort, this second route is easier and simpler. At a code level, the only main differentiator is the culture code being requested. Building a multi-language website following this path will be very similar to building a normal single-language site, just with a few added extras.

The majority of the methods exposed by the Umbraco content-focused APIs take an optional culture code. This means that getting different language content is as simple as supplying that optional parameter within your controller code.

Things become a little more complex when you need to structure the content using multiple homepages within the page tree. Instead of simply changing the culture code, you need to redirect users to a different area within your website.

When you need to perform redirection like this, you will also need to consider the routing and redirecting rules in more detail. Some questions you might want to consider include:

- How do you deal with links? For example, how do you prevent links from accidentally jumping a site visitor from one language to another one by accident?
- When someone firsts access the site on the naked domain, which homepage should load?
- Do you want to make use of culture codes as culture codes are not mandatory in this approach

Being able to answer these questions is key in order for you to optimally build a multi-language website. This is why understanding the content strategy is key to success.

Single-instance architecture

If your content strategy dictates that you can build your multi-language website as a single instance within the content tree, life should be pretty easy. The majority of your set-up tasks will be configuration tasks inside of the CMS rather than development tasks. Within this section, you will be taken through the entire process so you can get everything up and running quickly and easily.

When it comes to configuring the CMS, the first step is to enable the languages that you want to allow. After enabling some languages, you then need to enable multi-language mode in a variety of places. Enabling multi-language mode is done by setting the `Allow vary by culture` setting on all applicable document types.

After doing this the next step is to enable routing. Each language page will need a unique URL in order for your site visitors to access it. After doing all these things your pages should be accessible for each language you enabled.

Enabling languages

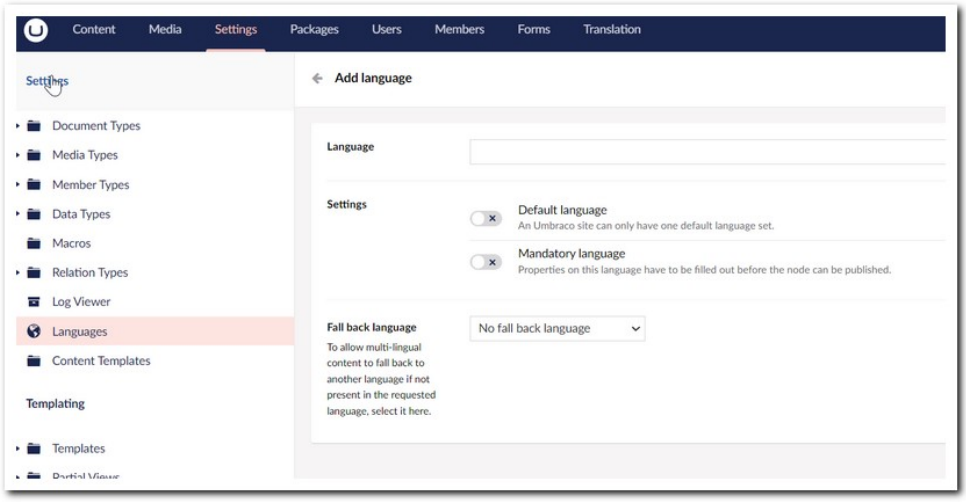
By default, Umbraco ships with a single language enabled which is usually English. If you want to work with additional languages you need to explicitly enable them within the CMS

You can enable languages from within the `Settings` section. After opening the settings page, within the left-hand sidebar, you should see a node called `Languages`.

From this page, you can enable a new language by clicking on the `Add Language` button. The modal that pops up will allow you to select the language that you want to enable.

For each language, you can decide if it will be the default language, or if it will be a mandatory language. Being able to specify if languages are optional or mandatory languages is very handy when publishing content. When you are trying to keep your website structure consistent being able to prevent a page from being published unless all variations have been filled in is a good guard.

When you create your English content, if French is a mandatory language, you will not be able to publish that content until the french version is added as well. This feature definitely helps to make it easier to maintain your project.



Adding a language

If you do not want to make a language mandatory-you can also specify a fallback language. The fallback language will be used whenever a translation is missing.

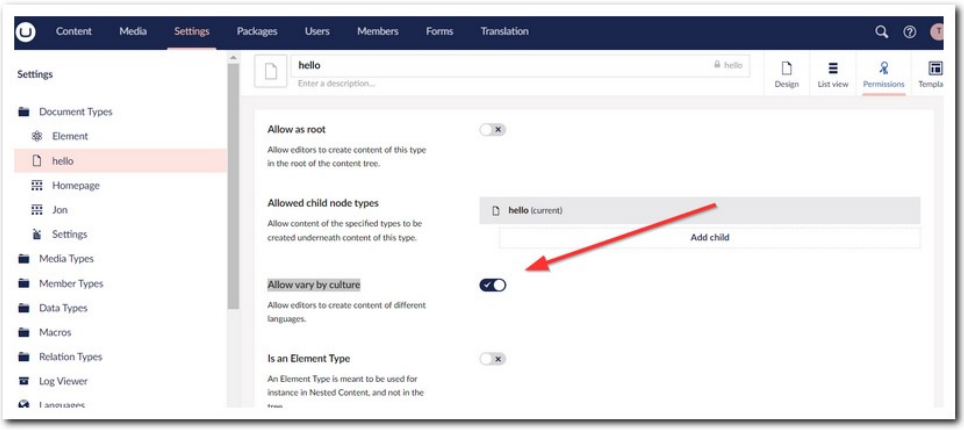
Enabling multi-language mode

After adding all the languages that you want to work with, you need to enable multi-language mode on all of your associated document-types. This is done from the `Document Type` screen within the `Settings` section.

By default multi-language mode is disabled on all document-type, which can definitely catch you out. If you enable a language and wonder why it's not working on a page, it's usually because you need to go into the associated document type definition and update some configuration.

There are two aspects of a document type that you need to update. You will need to enable multi-language on a page level AND at a per-property level. Enabling multi-language mode on a per-property basis is a bit tedious, however, it does give you a lot of flexibility.

To enable multi-language mode on a document type level, go into all of your document types and edit each one. Within the `Permissions` tab, you should see an option called 'Allow vary by culture':



Enabling Allow vary by culture on a document type

After enabling the document type to be used with multiple-languages, you will need to repeat the process on every property contained within that document type. For each property, click on the 'Settings' cog. From the property modal, enable 'Allow vary by culture' . Repeat this process for all document types and properties!

Routing

When you enable multi-language mode within the CMS, you will need to consider URL structure. Each multi-language page will need a unique URL. Without a unique URL, when a site visitor tries to access a page it will simply throw a 404 error.

By default, Umbraco is configured to work with only a single language. To allow pages to work with multiple languages, you will need to define a hostname per language. When it comes to multi-language URLs there are two strategies available to you, the sub-domain approach, and, the sub-directories approach:

Sub-domain strategy: When going the sub-domain route, you will add the language identifier as the first segment within your page URL structure, e.g. `uk.website.com`.

To enable a sub-domain strategy you will need to do some configuration outside of Umbraco. Specifically, you will need to add some new entries within your DNS provider. Within your DNS provider create a new A-record for each sub-domain. Point the root to your web server.

Sub-directory strategy: The alternative strategy is to use sub-directories. The URL structure within a sub-directory strategy looks like this`www.website.com/uk` or `www.website.com/fr`.

As the language segment comes after the main domain name, DNS changes are not required.

If you go down the multi-instance route, this will be the routing path you will have to take. In this strategy, the name of each homepage needs to map to the language code. The french homepage would be named `fr` and

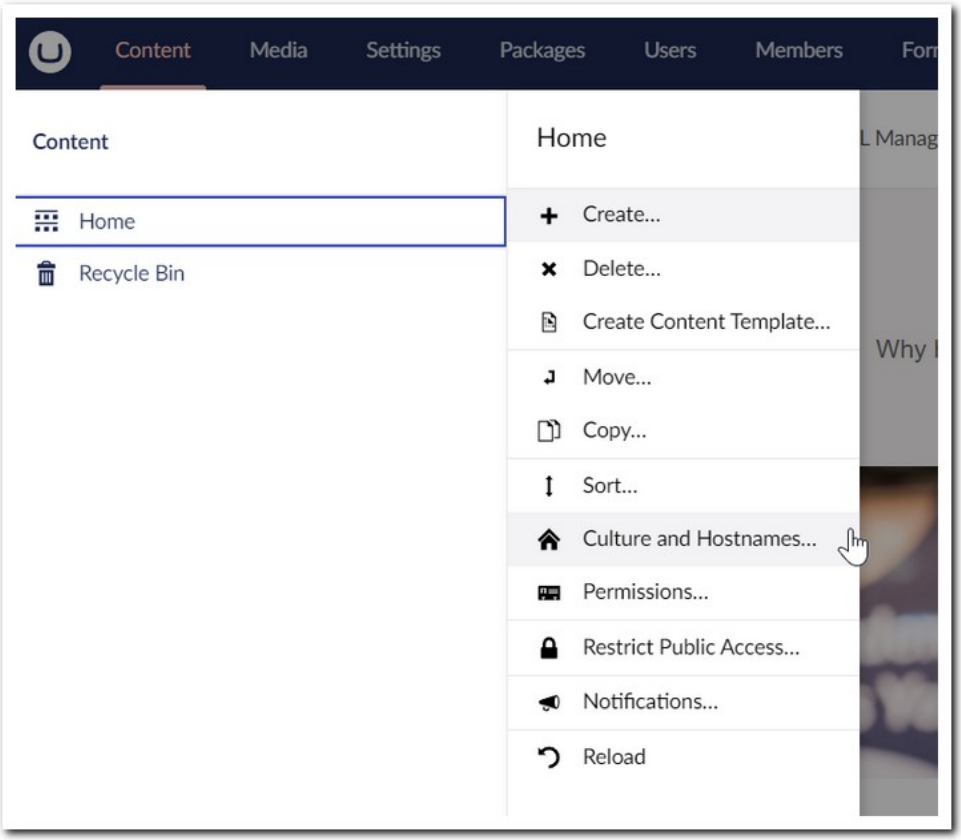
the English homepage would be named `en`. All routing will now magically work without you needing to do any DNS updates.

Be conscious that if you go this route in a single-instance architecture, you may need to write a composer and a content finder to make the routing work. It should be possible to get this to work, without having to do this though!

After deciding on a URL strategy, the next step is to map your hostnames within Umbraco. This is done from the content tree within the `Content` section. As a starting point, open the Umbraco context menu on your homepage. From this menu, you should see an option called `Culture and Hostnames`.

If you do not see this menu item within your Umbraco instances context menu, you may need to update the content editor viewing permissions. To do this open the context menu again on the homepage and select `Permissions`. From this tab make sure the editors/admin role has the `Culture and Hostnames` ticked.

The `Culture and Hostnames` page will allow you add one or more domains. For each domain you add, you need to map it to a language.



Setting the Umbraco culture

After mapping a domain to a language, whenever an incoming request is made to the server, the server will inspect the incoming URL and if a match is found the language will be automatically set for you. After Umbraco sets the language of the request, the APIs will automatically return

content in the correct language for you. Remember, the primary language is important for fallback content!

Adding content

After applying all these changes, within the page editing screen an additional dropdown menu should now appear within the pages title box. Switching the language here will allow an editor to add content for that language version:

Changing The Language

It is now a simple case of creating content and publishing it!

How To Create A Language Swapper

After creating all this amazing multi-language content, you will also need to provide a language picker. A language switcher will give your site visitor the power to decide how they interact with your website.

UX-wise, the language picker will typically live within the site's header, meaning it will be rendered from within your global layout. The picker will typically be a dropdown list, that allows the visitor to set their preferred language.

I will show you how to build a simple language picker, however, the code below is not the only route you can take. The first task is to get a list of all the enabled languages. You can do this by using the `Cultures` property exposed by the `CurrentPage` object within a Umbraco-powered controller. Iterating through this collection will give you a list of the enabled languages.

If you have set the hostnames correctly, each page will have a unique Url. Assuming you used a sub-domain strategy, the UK page Url structure might be `uk.website.com/blog-1` and the German one `de.website.com/blog-1`. To create a language swapper, all we need to do is render out a link for each culture on that page.

After a site visitor has been jumped to the German version of the page, all the page links will then automatically point to the German equivalents. In essence, the language picker can just be a collection of links. The code to

access all the associated cultures and their URLs in code is fairly straightforward:

```
public override IActionResult Index()
{
    var documentType = new MyDocumentType(CurrentPage, _publishedValueFallback);

    var languagePickerLinks = new List<LinkPoco>();
    foreach (var culture in blog.Cultures)
    {
        var link = new LinkPoco
        {
            Url = documentType.Url(culture.Value.Culture),
            Name = documentType.Value<string>("name", culture.Value.Culture),
            Culture = culture.Value.Culture
        };

        languagePickerLinks.Add(link);
    }
}
```

Creating a language swapper

After passing these links into the view, it is just a simple case of iterating through that list and rendering out a dropdown list or some other equivalent!

Multi-site architecture

The alternative way of building a multi-language website within Umbraco is to in essence create multiple websites in your single Umbraco instance.

In this architecture, instead of using the Umbraco switcher to switch between the different languages, you create multiple homepages within the content tree. The content editors will then create all the content and the pages for that language underneath the corresponding language homepage.

If you decide to go this route, the interesting thing is that technically you will still be building everything within Umbraco as a single-language website. On this path, you can ignore a lot of the steps mentioned in the last section. You do not need to add a second language from the `Lanaguges` section and you do not need to go through the hassle of updating all of your document types to enable the `allow vary by param`.

The reason why you can ignore these steps is due to the set-up. In the previous architecture, a single Umbraco node can have multiple variations. One variation per enabled language. To access a variation, you need to configure all that stuff to ensure multiple URLs can map to the same node. You do not bump into that limitation in this architecture.

In this architecture, it makes more sense to use the sub-category URL strategy, rather than use a sub-domain strategy. The page name of each homepage should mirror the language code it will represent.

Instead of simply calling the homepage `/`, you would call each one something like `en`, `fr`, or `de`. Using the language code as the page name means that the routing will automatically resolve for you. Following this architecture, your Url structure will look similar to this.

- www.website.com/en/
- www.website.com/fr/
- www.website.com/de/

As the language code will exist as a segment within the Url and within the backend, each node within the content tree will only contain a single variation. Routing will automatically resolve by page name alone.

The other benefit of multiple homepages is that content editors are free to organically grow content as they want. If you decide to go this route, the language switcher code will be slightly different. As each language could have a completely different folder structure, how you build the language switcher will differ compared to the last section.

In the previous switcher example, you could switch based on the current page object by rendering all of its variations. As we do not use variations in this approach, the best alternative is to use the homepages as the data source of the switcher. When we use the homepage, when someone switches the language you will be able to jump them to a page that you know will exist. On selection, the site visitor will be jumping to the language homepage.

```
var allLanguagesHomepages = UmbracoContext.Content.GetAtRoot().Where(x => x.ContentType.Alias == "myDocumentType");
foreach(var language in allLanguagesHomepages)
{
    var link = new LinkPoco
    {
        Url = language.UrlSegment(),
        Title = languageName,
    };
}
```

Multi-site language picker

Another consideration within this architecture is what happens when someone accesses the root domain? If someone accesses the naked domain, e.g. `/`, what will happen?

Typically, in these scenarios you have two options. One, you can automatically redirect the site visitor to the most appropriate homepage. Second, you create some type of landing page and they can choose which of the language pages they would like to visit. Most websites tend to pick option one. For example, when I go to `https://www.delonghi.com/` I am automatically redirected to `https://www.delonghi.com/en-gb`.

You could build out this capability within Umbraco in a few ways:

- You could create a custom `ContentFinder` and load the homepage that you want to load. SEO wise this is not ideal as you will end up with two pages with the same content.
- You could create a redirect rule, that performs a 301 redirect that jumps a site visitor to your default language.
- You could even implement something like the MaxMind's GeoLite2 free IP locator. This free service will tell you the site visitor's location, which you could then use to jump your user to the most appropriate homepage. You could add this GeoLite2 code within a component that loads for all naked domain requests.

Multi-language tips

At a code level, things change slightly when you enable multiple languages. You will still be using the same Umbraco APIs that you have been learning about throughout this book, however, you will need to start passing in at least one additional parameter into some of your existing Umbraco-related code.

After enabling multiple languages, do not be surprised if certain areas within your codebase suddenly break. The reason for this is due to variations.

When you have a single language, each page within the CMS is represented by a single variation. If you wanted to perform a save operation on any page, life is simple, as long as you have the correct page ID, the API knows exactly which variation you want to update.

After you enable multiple languages within Umbraco, each language variation is still represented by the same page ID, however, for each language, a new variation will be created within the database.

In order for the API to differentiate between the language variation that you would like to work with, you will need to pass in a language code. When Umbraco is armed with the page ID and a language code, it will know which page and variation you would like to work with. Without the language code, whenever the API tries to perform an operation on a page, it will get stuck. As the API will find multiple pages, without knowing the exact version it needs to pick, an exception will be thrown.

When it comes to this language identifier, Umbraco does not re-invent the wheel. Umbraco makes use of the standard .NET language code representation.

This representation is used by the CultureInfo class. The `CultureInfo` class specifies a unique name for each country and language. The culture code is typically made up of five characters, split into two parts. The first part of the code is the language identifier. This code is defined within the

ISO 639-1 standard. This code is then followed by a hyphen, followed by a two-letter country code. The country code is defined within another standard called the ISO 3166 standard.

Using this system, every country and language combination in the world can be uniquely identified. I am from England and I supposedly speak English, so my language code would be `en-en`. If I was from the US, my culture code would be `en-US`. Passing this code into certain Umbraco APIs will allow the API to update the specific version you want to work with.

You will notice that the majority of the methods exposed by the Umbraco APIs that query content, all take an additional optional parameter called `culture`. Whenever you see this optional `culture` parameter, this is where you pass in that language code.

The `culture` parameter is always optional. This is why after enabling multi-languages some of your code might break. It is very easy to write code within Umbraco that assumes that no language variations exist. An example of how to query the CMS using the `culture` parameter is shown below:

```
public HomepageController()
{
    UmbracoContext.Content.GetAtRoot("en-en")
}
```

Getting Language-Specific Content

In the example above, I am hard-coding the language code, which is not ideal. A far better approach is to access the site visitors' cultural preferences dynamically. Within .Net Core, there are a few options for accessing the language culture. Just like the .NET Framework, you can still use `CurrentUICulture` and `CurrentCulture`:

```
CultureInfo uiCultureInfo = Thread.CurrentThread.CurrentUICulture;
CultureInfo cultureInfo = Thread.CurrentThread.CurrentCulture;
```

Get language code using CultureInfo

As we are using .NET Core though, we should favour dependency injection. This code does the same, however, it will also ensure better testabil-

ity:

```
public class LoggedIn : Controller
{
    public LoggedIn(IRequestCultureFeature requestCultureFeature)
    {
        var culture = requestCultureFeature.RequestCulture.Culture;
    }
}
```

Get language code using CultureInfo

If you try to use this code, you may notice it does not work out of the box. When this happens, you will still need to add some additional configuration within `Startup.cs`

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        var supportedCultures = new List<CultureInfo>
        {
            new CultureInfo("en-EN")
        };

         var localizationOptions = new RequestLocalizationOptions
        {
            DefaultRequestCulture = new RequestCulture("en-EN"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };

        app.UseRequestLocalization(localizationOptions);
    }
}
```

Configuring Startup.cs

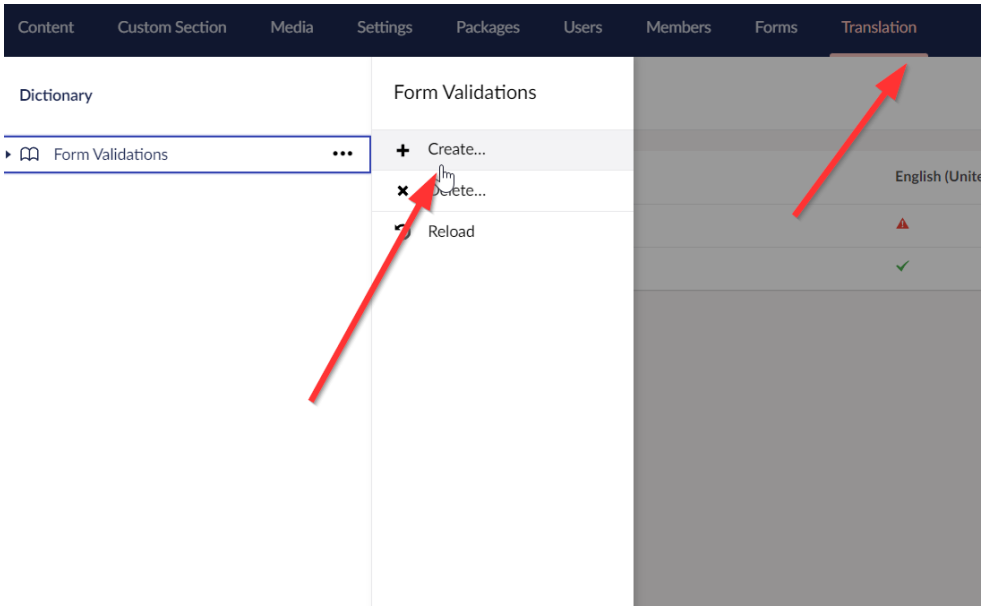
The code above should hopefully be pretty self-explanatory. Define which language codes you want to support. One interesting consideration from this code is the `DefaultRequestCulture`. When working on any .NET Core website you should always set a default language and the same is also true within Umbraco.

Translation Dictionary

Whenever I have worked on a multi-language project, typically, not all content is modelled within a document type. Often content like form validation messages, placeholder text, etc.. have no place to live. In many projects, developers cheat and hard-code these values within the view layer.

In true multi-language website, you can not get away with that. You will need a mechanism to deal with these outliers. When you find these oddballs, Umbraco provides the translation dictionary.

The translation dictionary can be access from inside of the CMS from the `Translation` tab. The translation dictionary is in essence a simple key/value store for content that takes into consideration language. Creating a translation inside of the CMS is simple enough. Open the context menu for the 'Dictionary' item in the left-hand sidebar and select the 'Create' option:



Create a new translation

The name that you give a translation will be the alias that you reference within your code. I recommend that you avoid using non-alphanumeric characters within your naming convention, just to make your life easier. For each translation, you will be able to add some text for each language you have enabled within the CMS.

To access a translation in code, you can use the `ICultureDictionaryFactory` API. The `ICultureDictionaryFactory` exposes a single method called `CreateDictionary()`. Despite its name, this method will give you access to the fully-populate Umbraco dictionary, which you can then query:

```
public MyController(ICultureDictionaryFactory dictionaryFactory)
{
    _dictionary = dictionaryFactory.CreateDictionary();
    var translatedText = _dictionary["translationAlias"];
}
```

Create New Member

You should always favour using `ICultureDictionaryFactory` whenever you need to read a translation value from the dictionary. If you want to perform more admin-related tasks to the dictionary, like creating a new translation in code, it is also possible to use `ILocalizationService`.

`ILocalizationService` is not as performant as `ICultureDictionaryFactory`, however, it will give you access to a lot more options around managing translations at a code level. Using `ILocalizationService` you will get access to method like `Delete()`, `GetDictionaryItemById()`, `GetLanguageById()`, `GetRootDictionaryItems()`, and `Save()`

```
public MyController(ILocalizationService localizationService)
{
    localizationService.Delete(new DictionaryItem("translationAlias");
    localizationService.GetAllLanguages();
}
```

ILocalizationService example

So the general rule of thumb around these APIs is that for read only usage favour `ICultureDictionaryFactory`, for more admin related translation tasks use `ILocalizationService`.

IVariationContextAccessor

The last API that is useful language related API to know about is `IVariationContextAccessor`. You know about the `UmbracoContext`, there is also a `VariationContext`. This is context defines the current culture. The umbraco APIs use this to return the correct content. Using `IVariationContextAccessor` is is possible tog et, and set the `VariationContext` yourself. Using `IVariationContextAccessor` will

allow you to temporarily set the culture for the current request. It can be applied like this:

```
public class MyController : Controller
{
    private readonly IVariationContextAccessor _variationContextAccessor;

    public MyTController(IVariationContextAccessor variationContextAccessor)
    {
        _variationContextAccessor = variationContextAccessor;
    }
    public ActionResult Index()
    {
        var culture = "en";
        _variationContextAccessor.VariationContext = new VariationContext(culture);
    }
}
```

IVariationContextAccessor example

Further Reading

- More information on CultureInfo
- The ISO 639-1 standard

Umbraco Good Practices

This final chapter will be a mix of knowledge that I think you will find useful for building website using Umbraco CMS. This chapter will cover everything from upgrading tips from Framework to Core, to caching and performance tips.

Upgrading Framework to Core

The focus of this book so far has been on implementing a brand new website using Umbraco and .NET Core. This book has assumed that you had the luxury of starting from a blank slate, however, in reality, this is often not the case. On a lot of projects, you will need to upgrade a website built using Umbraco v8 or below and convert it into either a Umbraco V9 or V10 website.

Upgrade projects are seldom fun and often daunting. One of the biggest challenges for people upgrading from a previous version of Umbraco to v9 or above is knowing where to start. When you do not know the steps that you need to take in order to successfully complete an upgrade project, life is not fun. When you find yourself faced with a million unknowns, writing down a plan will make the project seem a lot more manageable.

This section will walk you through a proven and battle-hardened upgrade plan. I have personally worked on over 50 upgrade projects during my career. Based on all these experiences, I have found this plan the easiest to implement. This plan is CMS agnostic and it can be used whenever you are faced with an upgrade project where the site's look and feel are not changing but the underlining technology is.

There are a variety of strategies for how you can upgrade a Umbraco website. A lot of tutorials online will recommend upgrading everything all at once and then fixing the symphony of resulting issues that ensue. I definitely do not recommend this approach which is also named the big bang release strategy.

If you try and upgrade all your code at the same time and compile the site, expect to see thousands of errors. Expect to spend a few days of effort simply getting the project to compile again.

When deciding upon an upgrade strategy, I recommend picking a plan that will allow you to move faster by taking constant little steps. Moving one step at a time will make your life much easier, so how do you move slower?

My goto upgrade plan involves creating a new vanilla website in the version of your choosing and then copying the code from the old solution into the new solution. Instead of simply copying all the code at the same time, you copy just enough code to get a single page working. When pick-

ing which page to start with, I recommend you pick your simplest page, something like a content page rather than the homepage.

The idea behind the methodology is that you want to get the easiest thing working and compiling quickly. This means getting the page to render minus the header and footer. As getting something to render as quickly as possible is the goal, you should omit any references to the layout file from the page's corresponding view. Removing this reference will mean that you will not need to copy any of the code required to generate the header and footer at the beginning, simplifying your initial tasks.

After you have successfully upgraded all the code required to get that page working, the next step is to re-add the reference to the header and footer. As part of enabling the layout, you will need to copy any related code from the old project to the new one. The benefit of this incremental plan is that it will reduce the amount of code you need to copy per step. you will find that the smaller the chunks you need to copy, the more manageable the task will become.

After you have gotten that page working, pick another page and repeat the process. Eventually, you will have most of your website upgraded. This focus on getting one page at a time working will require a little more admin work, however, the benefit is that you can get something working much quicker. It's this ability of constant continual progress that makes the upgrade process that much quicker.

After you have all of the webpages working within the new solution, you can then focus your attention on getting any back-office extensions, redirects, routing rules, sitemaps, RSS feeds, APIs, etc.. working. The advice when upgrading these other tasks is the same. Pick one capability, only copy the code related to that feature within a single step. Fix the feature and move on to the next. Eventually, your upgrade will be completed!

The Plan

This list is an overall summary of my upgrade plan. The first part is to make sure you have all the prerequisites covered. You might be tempted to skip these steps, but don't. Failing to adhere to these steps will simply waste your time as the upgrade will not complete successfully without them!

- Take a backup of your Umbraco database and web files
- Ensure you have the correct version of .NET installed
- Ensure you have the correct version of Visual Studio installed and patched to the correct version
- Ensure you have SQL Server and SSMS installed

Next, you need to create a blank vanilla Umbraco site using the version that you would like to upgrade to:

- Create a blank database for your new Umbraco website

- Go through a brand new Umbraco installation using the version that you would like to upgrade to. Create a vanilla Umbraco build.
- Make sure the namespaces match your existing project
- Create any class libraries with the solution directory and set them to use either .NET 5/6

After this you can upgrade your existing database to the latest version:

- Restore your old Umbraco database onto of the database created above.
- Open the websites `appsettings.json`. Change the `umbracoDbDSN` connection string to target the new database you created above, e.g. `umbracoDbDSN": "Server=.;Database=DocsSite;Integrated Security=true"`
- Re-run your website in a browser and you will now be presented with the Umbraco upgrade wizard progress
- Follow the steps to upgrade the database.
- After successfully upgrading try to log into the backend (`/umbraco`) with your admin account. If you encounter any issues see the troubleshooting section below. Sometimes the password might not upgrade correctly so fix that before moving on
- After logging into the backend, regenerate Umbraco models builder and check the health status page. You will likely need to add some config with `appsettings.json` to get this working (steps here)

You have now come to the hard part, updating all the code to work. This part is the most time intensive part:

- Pick a simple page on your site and get it working
- Use a diff tool like WinMerge to copy the related code from the old solution into the new one
- Comment out any layout references within the page's view file
- Compile the solution and fix bugs. This step will involve a lot of re-factoring.
- When the page is finally working, add the reference to the layout back within the view
- Copy all the code related to generating the header, footer etc… from the old site to the new site.
- Compile, test, refactor, and fix the issues
- After the page and header and footer are working, repeat the process. Pick another page, rinse and repeat
- After getting all the pages working, add any caching related code to the solution

After you have all the pages working, the last group of tasks is around getting everything else working:

- Add deployment transforms
- Backoffice extensions

- Redirects
- RoutingInitialization
- Scheduled tasks
- Robots.txt
- Sitemap.xml
- RSS filteredPages
- Emails

After following all these steps, you should now have a fully working upgraded Umbraco website.

Obviously, this plan on paper looks easy enough to implement. In real life, sadly, it will not be this easy. Expect to bump into lots of random code bugs. Things that once existed will be no more. With the introduction of things like view components and a change in DI containers, expect the possibility of completely refactoring and re-architecture how various bits of your codebase works.

There are tools like the Microsoft Upgrade Assistant which can help upgrade a project from .NET Standard to .Net 5/6, however, as a lot of the underlining Umbraco APIs have changed this tool will not be as useful as you might hope. The Microsoft upgrade assistant will result in everything breaking at once, making it a complete ball ache to try and get everything back up and running.

Password Issues

If your site started life as a v7 website, you may encounter password issues after upgrading. The reason why this issue can occur is because of a change to how passwords were salted between v7 and v8. After you run the upgrade wizard, if you cannot log into the CMS anymore, you may need to reset the admin password.

There are two ways of resetting the admin password, via email or manually. To go the email route, you will need access to a valid SMTP server. After adding your SMTP details within `appsettings.json` you can then perform a forgotten password request from the Umbraco login page. You can then use this config to add SMTP details within `appsettings.json`:

```
"Umbraco": {
    "CMS": {
      "Security": {
        "AllowPasswordReset": true
      },
       "Smtp": {
          "From": "from",
          "Host": "host",
          "Username": "username",
          "Password": "password"
        }
    }
}
```

Adding SMTP details

The alternative way of resetting the password is the path that I prefer. In this route, we will make use of the Umbraco installer to reset the password. To trigger the installer to load instead of your website, remove your database connection string details from within `appsettings.json`:

```
"ConnectionStrings": {
    "umbracoDbDSN": "
}
```

Setting a ConnectionStrings

After saving the file, re-run your site and the Umbraco installer will now load. Add in your new admin username and password details. These new details will be the ones that you will use to log into the backend. Continue through the wizard and click on the Custom installation option when you are presented with it. Pick SQL Server as the database type and then re-add your existing database connection string details. Finish running through the installation process. The installer will reset your password

and as long as your database is already upgraded, nothing else will happen to the database:

Upgrading Considerations And Coding Issues

The version of Umbraco you had to upgrade from will determine how much code you will need to refactor. In terms of effort, the further back you need to upgrade from, the more code you will need to refactor. Moving from Umbraco v9 to v10 will be less painful compared to v8 to v9, while v7 to v10 will be the most painful.

The reason why moving from v8 to v9 is less painful compared to v7 to v9 is due to architecture. Within v8 many new concepts were introduced that are used within v9 and onwards. Within v8, dependency injection was introduced and many of the core Umbraco APIs switched to the style now used throughout v9 and above. Components and composers were also introduced for developers to hook into the pipeline, meaning less code within `global.ascx`. If your solution makes use of these newer features, your life will be slightly easier when you attempt to upgrade, however, you will still have a lot to refactor.

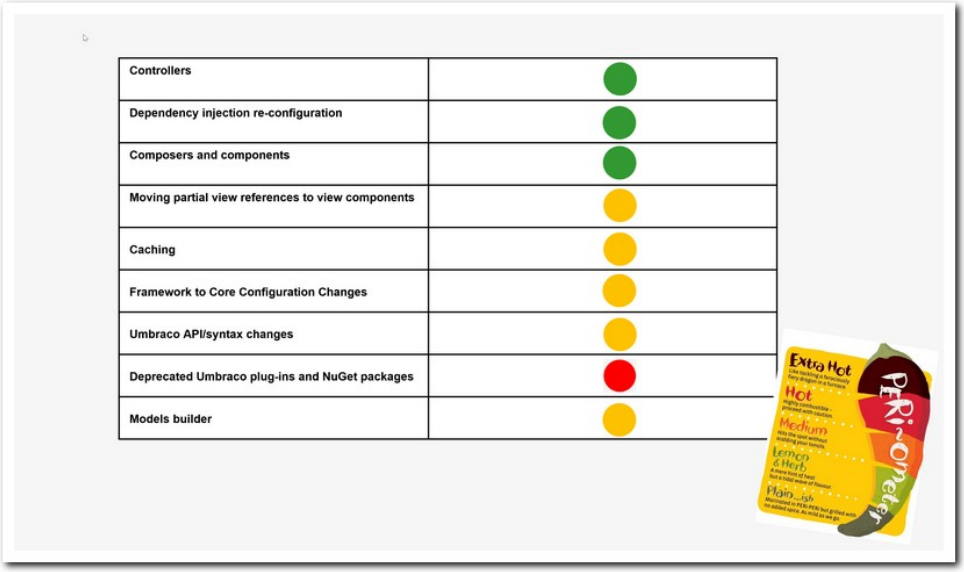
If you are on v7, you may want to consider a complete new build rather than an upgrade as the changes in architecture are substantial. For instance, all the code within your `global.asax` will have to be completely rewritten. The official recommendation is to go from Umbraco v7 to v8, before attempting a v9 upgrade. Going this prolonged route is probably longer than a complete rebuild, so you may want to consider your options before jumping in. Going with a new build will likely be a less frustrating and quicker route.

When upgrading, a lot of your existing code will break. Expect issues everywhere, especially in:

- Controllers (minimal)
- Dependency injection re-configuration (minimal)
- Composers and components (minimal)
- Moving partial view references to view components (medium)
- Caching (no output cache within .NET Core out the box)
- Configuration changes due to the move from `web.config` to `appsettings.json`
- Configuration changes due to the move from `global.ascx` to `startup.cs`
- Syntax changes
- Models builder modes have been renamed
- Replacing deprecated Umbraco plug-ins
- Replacing deprecated NuGet packages

The largest investment of time when upgrading will be the refactoring effort. Making .NET Framework code work in .NET Core is not a one-button click task. Your refactoring challenges will range from small syntax updates to big architectural changes.

The .NET Framework relied heavily on `System.Web` which has now gone. This means you will likely need to refactor a lot of things. For awareness, below highlights a high-level overview of areas you will need to refactor and a T-Shirt size of the effort you will need to invest in each area:



Umbraco Upgrade Effort

When it comes to tackling each area, expect to make smaller refactoring such as:

- Changing all code that references `HttpContext.Current`
- Replacing any code that uses older Umbraco services and APIs to use the new dependency injection-friendly APIs
- Refactoring properties to methods , e.g. `WriterName` to `WriterName()` and `Url` to `Url()`
- Fixing deprecated property types. Within .NET Core, the `IHtmlString` interface has gone. A lot of the out-of-the-box Umbraco properties like Markdown and RTE were of type `IHtmlString`. You will need to update any code that references this type and also test that after your refactoring effort HTML is still being output rather than plain text. I opted to use a new type supplied by Umbraco called `IHtmlEncodedString` as a replacement of `IHtmlString`. To render HTML using `IHtmlEncodedString` you will also need to render the property using `Html.Raw()`. This is not ideal, however, it works.

Unfortunately, there is no magic silver bullet when upgrading. The process is not technically complex, however, it is time-consuming. Like normal refactoring, keep changes small and frequent. Do not be scared to undo changes if things are not working out. Frequent Git stashes are your friend!

Upgrading Umbraco V9 to Umbraco V10

Umbraco 10 was released in June 2022. The biggest difference between V9 and v10 is that Umbraco V9 is ASP.NET 5 and Umbraco v10 is ASP.NET 6.

If you were not aware, Microsoft has made some pretty significant performance improvements in .NET 6. .NET 6 is meant to run around 40% quicker compared to .NET 5. If you are on V9 and want to improve performance, one of the biggest wins you can make is to simply upgrade to .NET 6.

The upgrade complexity from jumping between Umbraco V9 and v10 is significantly easier compared to the v8 to V9 path, however, it will not be all plain sailing. There are a few challenges that you will need to solve in this upgrade path.

In .NET 6 `startup.cs` has been deprecated. Within V10 there are also some breaking code changes. If you decide to upgrade to v10, expect that some of your code will break. The impact of these changes will be minimal. You should be able to fix any broken code by applying a small refactoring. If you are lucky, the whole upgrade process should only take you a few hours at max.

In order to successfully upgrade your website, you will need to apply certain changes. These are:

- Convert your website library and class libraries to .NET 6
- Bump your Nuget Umbraco packages
- Bump Your .NET packages
- Build and fix broken code
- Decide how to deal with the Program/Startup change
- Upgrade the database
- Clear the cache

Below will cover each of these steps in more detail.

Bump Your Class Libraries To .NET 6: The first step on your upgrade journey is to bump your solution to use .NET 6. Without doing this, you will not be able to upgrade any of the NuGet packages. To upgrade your website to .NET 6 you will need to open up your website's `.csproj` file. For a .NET 5 site, the config within `.csproj` will look like this:

```
<PropertyGroup>
  <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
```

.NET Project configuration

To upgrade your site to version 5, update it like this:

```
<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
</PropertyGroup>
```

.NET Project configuration

As well as converting your website project files to .NET 6 you will also need to do the same for all of your class libraries as well.

Bump your Umbraco packages: After converting your project to .NET 6, the next step is to upgrade all of the Umbraco NuGet packages. I find upgrading packages way easier using NuGet package explorer compared to the terminal, but do what makes you happy. To successfully upgrade Umbraco you will need to bump the following packages to use a 10+ version:

- Umbraco.CMS
- Umbraco.CMS.Core
- Umbraco.Cms.Web.Common
- Umbraco.Cms.Web.Website

When it comes to upgrading these packages, my advice is to upgrade things on a per-project basis. The alternative path is to try and upgrade everything all at once at the solution level. This blaze of glory approach was the path I tried first and it didn't go well. I encountered lots of versioning conflicts. I found life to be way easier when I upgraded the smallest and least referenced class library first. After successfully upgrading one library, you can then rinse and repeat from the least used library to the most used. Finishing with your website application last.

Bump Your .NET Packages: After upgrading Umbraco, you will also need to do the same for a few base .NET packages. Within my solution, the two

.NET packages that I needed to upgrade were:

- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.Json

If your solution uses either of these packages and you forget to upgrade them, expect an `appsettings.config` related error and the site to fall over when you try to boot it up.

Fix Broken Code: After upgrading all of the relevant Nuget packages build the solution. If you are lucky the site will compile OK. If you are not some of your code will break. The reason for this is that there are a few breaking changes in v10.

There is an official list created by Umbraco about all these the breaking changes that you can find here

On my project, I was using the `ContentSchedule` property on `IContent`. This property was deprecated in v10. To get the same information, I needed to inject the `IContentService` API and then use `GetContentScheduleByContentId()` method. This refactoring is shown below:

```
IContent _content;
var scheduledDate = _content.FullSchedule.FirstOrDefault(x => x.Action == ContentScheduleAction.Release);

// New Code
IContentService _contentService;
var scheduledDate = _contentService.GetContentScheduleByContentId(node.Id)
                                   .FullSchedule
                                   .FirstOrDefault(x => x.Action == ContentScheduleAction.Release);
```

GetContentScheduleByContentId()

When compiling my website project, I also encountered a duplicate file issues. This was because I included the `Umbraco` folder in my solution files. To fix the issue I had to `Exclude From Project` my `Umbraco` folder within `wwwRoot`

Program/Startup: The biggest pain in the ass with the jump to .NET 6 is that `Program.cs` and `Startup.cs` has been merged. .NET 6 also uses something known as “minimal hosting” and “minimal APIs”. This means that within `Program.cs` you do not add a namespace or class declaration. You can just write code. As a long-time .NET developer, this change feels very odd.

One good thing about Microsoft is that they always support legacy ways of working. To prove this point, try installing a Windows 3 application on Windows 10! Following this tradition, it is still possible to carry on using `Startup.cs` in .NET 6 and personally, I prefer this separation of concerns. Having one giant file is not as optimal, as being able to split my config down.

How this change will impact you, is that when you run your website, you will likely encounter this exception before your site boots-up:

```
System.ArgumentNullException: 'Value cannot be null. Arg_ParamName_Name'
```

To fix this issue, I left my `Startup.cs` untouched, however, I updated my `Program.cs` to look like this:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

var builder = Host.CreateDefaultBuilder()
    .ConfigureUmbracoDefaults()
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStaticWebAssets();
        webBuilder.UseStartup<Startup>();
    })
    .ConfigureLogging(x => x.ClearProviders())
    .ConfigureAppConfiguration((ctx, builder) =>
    {
        builder.AddJsonFile("appsettings.json", false, true);
    });


var host = builder.Build();
host.Run();
```

GWorking Program.cs example

Upgrade The Database: After the solution builds, the next step is to upgrade the database. This is done by running the Umbraco upgrade wizard. To access the wizard, run your site in a browser and it should automatically load. You will be asked to log into Umbraco, Log-in and click the upgrade button. If the upgrade goes according to plan, the CMS should load. After you successfully log into the editor, you should do two things before testing your website works:

Regenerate the Umbraco model builder models: You should generate the Umbraco model builder models before running your site. This ensures the generation process is still working and that all the document types have the correct properties. To re-generate the models, log into the backend and go to this screen:

`Settings` => `Settings` => `ModelsBuilder`

Click the `Generate` button:

Regenerate Examine indexes: You should also regenerate your Examine indexes. Make sure you regenerate each of the default indexes:

- ExternalIndex
- InternalIndex
- MembersIndex

When upgrading I encountered a few warnings in the backend. To fix these issues, I turned off my web server, performed a `Clean` build within Visual Studio and then reloaded the site. This fixed those errors. If you encounter odd issue I recommend you try the same.

The upgrade process on my project took me a little longer than I thought, however, it was done and dusted in a few hours. The biggest pain were the breaking changes and the change in `Program.cs` and `Startup.cs`.

Performance

Within this last section, you will learn how to configure your Umbraco-powered application to improve page load times. No matter how badass your website looks, if your pages do not load quickly enough your customers will get fed up and go elsewhere.

How you apply performance optimization tweaks within .NET Core differs greatly compared to .NET Framework. With the move from `global.ascx` and `web.config`, you will not be able to simply port any of your existing performance-related code into a .NET Core project.

Asides from configuration, more fundamental things have changed as well. I found that one of the most useful performance optimization tools a developer could make use of within .NET Framework was the output cache. When enabled, the output cache will store the output of a rendered page within an in-memory cache. The output cache no longer ships out-the-box with .NET Core. Within Umbraco 9 and above, if you want to implement a similar caching technique you will no longer be able to use the Framework and you will need to use an alternative!

It is not just the output cache that has changed. A more Umbraco-specific change is the departure of the ClientDependency Framework. Within Umbraco v8 and below, a developer could make use of the ClientDependency Framework to perform CSS and JS bundling and modification at build time. If you want to implement bundling and modification using Umbraco 9 features, you now need to use a new framework called Smidge. There are too many changes between Framework and Core as well as API changes between Umbraco V8 and V9 to list every single one of them here. The takeaway is that you will need to re-learn how to do performance within Umbraco 9.

In terms of planning, performance tweaking should be one of the last tasks you undertake before going live on a solution. Within your project plan, I suggest you block out a week or two at the end of the project just

to focus on getting the page speed to an acceptable level. A lot of teams are tempted to try and implement performance tweaks as they go along during the build. Unfortunately, despite these good intentions, Studies have found that this type of premature optimization can counter-intuitively negatively impact performance. My experience mirrored this. Starting performance early tends to waste more of your time rather than save it!

In my Umbraco 8 book, I listed all the tools and techniques that developers can use to performance test a website. All those tips still apply to Umbraco 9. It is the implementation details that have changed rather than how you test page speed.

When it comes to improving performance within your solution, there are several different types of caching that you will want to apply. These include:

- Page-level caching
- View-level caching
- Minification and bundling
- Asset-level caching
- Object-level caching

Having a strategy that targets all of these areas will ensure your pages load quickly. Within this section, I will explain how you can implement each type of caching strategy.

Performance advice can never be a one-size-fits-all solution. This section is not meant to be a definitive guide on all things performance within .NET Core. For each type of caching technique, I will give you the basics, however, you will likely need to perform some additional research to get the performance just right for your solution.

For example, I have skipped considerations like distributed caching, CDN caching, managing a cache with authenticated or personalized content, doughnut caching, and bundling and minification at a Front-end level. If I have not listed something you think is important here, do not assume that's because your idea is a bad practice. It is better to assume I left it out simply because the topic of performance is simply too big to cover everything in detail!

When thinking about performance, the first question most developers ask is where to start? Should I implement a client-side cache or a server-side cache? The answer to this is both.

Client-side Caching: In this first caching technique, you will configure your server to add certain cache headers to incoming page requests. When set, these headers will tell your website visitor's browser to cache your pages after it downloads them for the first time.

After a site visitor has visited a page on your site, all subsequent access to that page will use the browsers' locally cached version. As you would expect, this type of caching can drastically reduce the number of calls made to your server.

The steps to enable client-side caching within .NET Core are very similar to most of the other techniques you will read about in this section. You will need to enable some middleware within `Startup.cs` (or just `Program.cs in .NET 6) and add some cache-specific headers to all requests that you want to cache. Below shows an example of how a cache header would look:

```
cache-control: max-age=90
```

This specific header will tell the browser to cache the response for 90 seconds. After 90 seconds have elapsed, on subsequent visits the web browser will request a new version of the page from the server rather than use its local copy.

When enabling client-side caching, determining the optimal cache time for a page can be very tricky and will need some consideration. After a browser has a cached version of a page, it will not re-request that page from your server until the cache duration has been met. There is no way to force the client's browser to re-call your server after a page is cached. Once cached, the only way for a client to see any content updates is to wait. If you enabled a one-year cache on your homepage, no matter how many updates you make to it, the client will not see those changes until they either clear their local browser cache or the cache duration has expired.

Caching pages that change frequently with a very long cache duration will mean your site visitors might miss out on important news and content updates. This is why when you use client-side caching, you will need to consider how often your pages update. Typically, you will want to set an expiry time somewhere between a few hours to a few days. To enable client-side caching the first step is to enable some middleware within `Startup.cs` for .NET 5 (or `Program.cs` for .NET 6):

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddResponseCaching();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseResponseCaching();
    }
}
```

Enabling Middleware

After enabling the middleware your next step is to define how long the browser should cache each page. You will need to apply these settings on

a per request basis, meaning you will need to add some code at the controller level. You will need to decorate all of the page controllers that you want to be cached with the `ResponseCache` attribute.

```
[ResponseCache(CacheProfileName = "Weekly")]
public class MyController : RenderController
{
    [OutputCache(Profile = "default")]
    public override IActionResult Index()
    {
        return View("~/Views/MyController/index.cshtml");
    }
}
```

Applying the ResponseCache on a controller

To use this attribute, you may need to install the Microsoft.AspNetCore.ResponseCaching NuGet package. The `ResponseCache` attribute will allow you to determine what gets cached and for how long. You can configure how the page is cached through six very useful properties:

- Duration
- CacheProfileName
- Location
- NoStore
- VaryByQueryKeys
- VaryByHeader

For the bare-bones cache configuration, you need to define the cache expiry time. This is done by specifying either a cache duration or a cache profile. For best practice, you should favour using profiles. Without using a profile, you will have to duplicate your cache settings within your controllers. This will violate the DRY principle which is not ideal. You can set up a cache profile within the middleware like this:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc(options =>
        {
            options.CacheProfiles.Add("Weekly", new CacheProfile()
            {
                Duration = 60 * 60 * 24 * 7
            });
        });
    }
}
```

Setting a cache profile using AddMvc()

If you look online, you will likely see an alternative way to enable the same config using `AddControllers()` like this:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        options.CacheProfiles.Add("Weekly", new CacheProfile()
        {
            Duration = 60 * 60 * 24 * 7 // 7 days
        });
    });
}
```

Setting a cache profile using AddControllers()

I do not think there's a big technical difference between the two, use the approach that works with your code. There is a downside with both examples above which is the profile settings are hardcoded. A better alternative is to define your cache profiles within config and dynamically read them in. This will mean you can change profiles on a per environment basis easily. You can do this by making use of `appsettings.json`. To define a cache profile within `appsettings.json` you could add this JSON:

```
using WebEssentials.AspNetCore.OutputCaching;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddOutputCaching(options =>
        {
            options.Profiles["default"] = new OutputCacheProfile
            {
                Duration = 60 * 60 * 24 * 7 // 7 days
            };
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
            app.UseOutputCaching();
    }
}
```

Adding a cache profile to appsettings.config

With your cache profile defined, you can access these settings by adding this code within `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddMvc(options =>

    var cacheProfiles = builder.Configuration
            .GetSection("CacheProfiles")
            .GetChildren()

    foreach (var cacheProfile in cacheProfiles)
    {
        options.CacheProfiles.Add(cacheProfile.Key, cacheProfile.Get<CacheProfile>());
    }
});
```

Defining cache profiles from appsetting.config

Finally, with your profile defined, you should also consider how your pages can be rendered. If the page can be rendered in different states, you may need to cache multiple versions of that page. For example, if you use personalization you will not want any personalized content to be cached.

Whether it be changing content based on anonymous/member content, personalized content, or whatever else you will need to add additional configuration. When you need to consider caching and page versions, I recommend that you research the other properties exposed by the `ResponseCache` attribute. By tweaking these settings you can get a very fine-grain level of control over how the page is cached.

On each page request, the middleware takes the cache duration defined within the `ResponseCache` attribute from the controller and add the cor-

rect headers. The response cache will also respect the `VaryByHeader` and `VaryByHeader` options, allowing for additional versions of a page to bypass the cache.

Server-side Caching: Traditionally, the quickest way to enable server-side caching within .NET Framework was to enable the output cache. Unfortunately, life is not so easy when it comes to .NET Core development. As a .NET Core application can run on any environment, having a single in-memory cache that works between Mac, Windows, or Linux is not as straightforward as it used to be. Instead, to get a similar capability within .NET Core you will need to turn to a third-party package.

The package that I have successfully used on my projects is called [WebEssentials.AspNetCore.OutputCaching](). The WebEssentials AspNetCore.OutputCaching package was written by Mads Kristensen, a dude who has written a lot of useful .NET utilities.

To enable the output cache you will need to enable the output cache middleware within `Startup.cs` (or `Program.cs` for .NET 6), like this:

```
using WebEssentials.AspNetCore.OutputCaching;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddOutputCaching(options =>
        {
            options.Profiles["default"] = new OutputCacheProfile
            {
                Duration = 60 * 60 * 24 * 7 // 7 days
            };
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
            app.UseOutputCaching();
    }
}
```

Enabling OutputCacheProfile

Once enabled, this middleware will read the cache duration set within the responses cache-control header. The way in which you add the cache headers to a request is the same as in client-side testing. You will need to decorate your controllers (or actions) with the `OutputCache` attribute:

```
[OutputCache(Profile = "default")]
public class BlogController : RenderController
{
}
```

Enabling OutputCacheProfile

After decorating your controllers with the `OutputCache` attribute, at run-time, when a page request is made and the corresponding controller is triggered, the WebEssentials output cache plug-in will read the controller's cache configuration and cache the page appropriately. The `OutputCache` attribute also works based on the cache expiration time. You will need to define either a `Duration` or a `Profile`. Profiles are defined within `Startup.cs` (or `Program.cs` for .NET 6). The process is very similar to the client-side process!

View-level Caching

To ensure that a page's HTML is generated as quickly as possible, you should try to cache as much of the page creation process as possible. As mentioned at the start of the book, within .NET Core a view can be broken into smaller chunks using something called a view component. One reason why view components improve page load time is due to their asynchronous nature. When used on a page, the server can parse multiple view components in parallel making the creation process slightly quicker.

View components can also be cached. Instead of making the server have to completely regenerate the contents of a view component each time, caching its output will further speed things up.

Caching view components is possible by using the caching Tag helper. Caching a view component using this helper is pretty simple, just wrap the components tag deceleration using the `cache` tag helper like this:

```
<cache expires-after="@TimeSpan.FromMinutes(5)">
    @await Component.InvokeAsync("Categories")
</cache>
```

Umbraco Upgrade Effort

The cache tag helper will also allow you to vary what gets cached by many different criteria including header values (`vary-by-header`), query-string value (`vary-by-query`), routing (`vary-by-route`), cookie (`vary-by-cookie`), and user (`vary-by-user`).

```
<cache expires-sliding="@TimeSpan.FromHours(1)" vary-by-cookie="my-cookie">
    <MyComponent />
</cache>
```

Vary By Cookie

A complete list of how the tag helper can be configured can be found here

On your project I recommend you add your site's header and footer into their own view components. Caching both of these view components is an easy win that will improve performance on all pages!

Asset level Caching

Web pages are comprised of more than just HTML. When creating a page, content editors will also likely want to enhance the user experience by linking to images, eBooks, videos, white papers, webinars and a whole host of other interactive content.

Simply uploading images and files within the Umbraco media library will not cache them. You will need to apply some additional asset-level caching configuration yourself.

Failing to add this configuration will mean you will encounter two performance niggles. The first issue will be at the CDN level (assuming that you are using one). When your assets are not set with the correct cache headers, your CDN will route all traffic back to origin. The consequence of this is that all your site visitors from around the globe will need to route back to wherever your server is located. Adding additional distance to a request will result in slower page load times.

The second issue is around capacity. As your customers will be making more requests back to your server, that server's total throughput will be reduced. The busier your server becomes, the less traffic will be able to access your site.

To improve performance you should enable asset caching. To do this, you will again need to enable some middleware and add some configuration within `Startup.cs`.

Before you add this cache configuration, you need to consider how long you want that asset to be cached. The good thing with asset caching is that determining how long to cache an asset is usually much easier. After an image gets uploaded and cropped it is highly unlikely that the asset will ever change. If it does, it is much more likely that a new image will be uploaded and linked to instead. This means that when applying caching onto static assets you can usually use a much high cache expiration time compared to page-level caching.

Using a technique called cache-busting, you can even set the cache duration to infinity if you really wanted to. In cache busting, a postfix is appended onto the assets URL. This cache identifier is usually calculated based on the asset's publish time and date. Updating the page HTML to point to the original asset with a new identifier will force the browser to re-fetch the asset from the server.

Cache busting is a topic that all web developers need to understand. If this topic is new to you, I suggest you spend some time researching it. A good tutorial to learn more about this subject can be found within the further reading section. Combining smaller page-level cache times with cache busting and extremely long asset cache times will give you the best performance results.

You can enable asset-level caching using the `StaticFileMiddleware` within `Startup.cs`. Just like page caching, when an incoming request for an asset is made, the middleware will add the correct caching headers to the response.

You should enable the `StaticFileMiddleware` early on within `Configure()` by making a call to `AddStaticFiles()`. This configuration needs to be at the top of the method because adding it towards the end of `Configure()` can result in assets not being cached correctly:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

UseStaticFiles()

After adding this configuration to your application, images and files added from within the Umbraco by the content editors should be magically served with the correct cache header.

This means that items added within the `media` folder within `wwwroot` should have the correct cache headers added to them. To check this within a browser view an image and check the header contains a value that looks like this `cache-control: public, max-age=432000`

Minification and Bundling

After making sure your assets load quickly, the next step is to ensure that all your site's CSS and Javascript files load as optimally as possible. To do this you will want to introduce some flavour of minification and bundling capabilities into your build process. Minification and bundling will ensure that your CSS and Javascript files are as small as possible.

There are a number of different ways to enable bundling and minification. Depending on your front-end developers' preference, your CSS and Javascript bundling conversion might be better suited to being triggered from within a node-based project using something like Vite or Web Pack.

In some projects, it might make more sense to do the bundling and minification server-side. The good news is that this is all possible out-of-the-box with Umbraco. When you originally installed Umbraco, behind-the-scenes an additional third-party NuGet package called Smidge was also installed.

Smidge is CSS and JavaScript file minification, combination, compression and management library for ASP.Net Core. Written by Shannon Deminick, Smidge is used by the Umbraco backend to improve page load time. As the CMS references its own CSS and Javascript files, bundling these files helps improve response times for content editors!

Smidge can be used either within a vanilla .NET Core website or within a Umbraco-powered website. As there are different use-cases for how Smidge can be used, there are several different ways that it can be configured. Using the Umbraco route, you can configure how Smidge using the `RuntimeMinification` setting within `appsettings.json`:

RuntimeMinification

Smidge can be configured with three `CacheBuster` modes:

- Version: The cache will be busted when the version changes
- AppDomain: The cache will be busted when the server restarts
- Timestamp: The cache will be busted based on a timestamp

To include which Javascript or CSS files should be included within the bundling process, you can use this code:

```
@using Smidge
@{
  SmidgeHelper.CreateJsBundle("myJsBundle").RequiresJs("~/scripts/jquery.js");
  SmidgeHelper.CreateCssBundle("muCssBundle").RequiresCss("~/css/tailwind.min.css");
}
```

Add CSS and JS files into Smidge Bundles

You can then render the resulting Smidge JS and CSS bundle using this code:

```
@using Smidge
@{
  SmidgeHelper.CreateJsBundle("myJsBundle").RequiresJs("~/scripts/jquery.js");
  SmidgeHelper.CreateCssBundle("muCssBundle").RequiresCss("~/css/tailwind.min.css");
}
```

Render the end JSmidge JS and CSS bundle

Alternatively, you can also get access to the bundled files using this snippet:

```
@using Umbraco.Cms.Core.WebAssets
@inject IRuntimeMinifier runtimeMinifier

@await runtimeMinifier.RenderJsHereAsync("myJsBundle")
@await runtimeMinifier.RenderCssHereAsync("muCssBundle")
```

Alterative bundling code using runtime-minifier

Object-level caching

Asides from page-level and HTML-level caching, you should also cache the results of any expensive operations. Querying an API for data, reading data from disk, reading pricing data from a PIM, and accessing commonly used settings are all examples of things that might be better off cached.

Without adding object and data caching, your code will be running slower than it could be. If you are calling an API on demand, the chance of timeouts occurring, or, making the server unnecessarily busy also increases. By introducing caching around these calls, you can also increase your site's reliability.

During your website's page execution life-cycle, if you have code that is called frequently that also takes time to process, that code is a good contender to be added to the object cache.

Out-of-the-box, .NET Core provides two APIs to allow for caching, `IMemoryCache` and `IDistributedCache`. The difference between these two caching helpers is where the cached data will be stored. Using the in-memory cache helper will add the cache data within the application's server memory. Whereas within a distributed cache, data is stored within an external service that multiple servers can access.

Out of the two, in-memory caching is definitely the simpler cache to implement. The main downside of this type of caching is cache sizes. Since in-memory caching uses your server's RAM you should treat it as a scarce resource. If you max out your server's RAM, expect bad things to happen. A good practice is to limit the number of items you add to the cache and always use a cache size limit.

If you need to make use of a distributed cache, I would recommend that you also consider implementing something like Redis, or another NoSql option before just jumping into `IDistributedCache`. Using the distributed cache helper within .NET core is pretty well documented here.

The in-memory cache helper does rely on some middleware for it to work, however, this middleware should be automatically enabled when you install Umbraco. If you have issues getting and retrieving items from the cache, you might want to manually add that middleware within `Startup.cs` using the `AddMemoryCache()` activation method.

To add an item within the cache, you need to inject the `IMemoryCache` API into your code using dependency injection. To access a value from the cache, you can use the helpers `TryGetValue()` method. `TryGetValue()` takes a cache key and if a corresponding item in the cache exists, it will return the corresponding value back:

```
public class HomeController : RenderController
{
    private readonly IMemoryCache _memoryCache;

    public HomeController(IMemoryCache memoryCache)
    {
        _memoryCache = memoryCache;
    }

    public override IActionResult Index()
    {
        var cacheKey = "myKey";
        var cachedData = "";

        if (!_memoryCache.TryGetValue(cacheKey, out cachedData))
        {
            _memoryCache.Set(cacheKey, "addToCache");
        }
    }
}
```

Adding an item to cache using IMemoryCache

Adding items to the cache is done using the `Set()` method. When adding items to the cache, it is also possible to define when the item should be removed from the cache. Adding this cache duration is important to prevent the server's RAM from filling up. To add the cache expiration policy you need to pass in an additional options object into `Set()`:

```
var cacheExpiryOptions = new MemoryCacheEntryOptions
{
    AbsoluteExpiration = DateTime.Now.AddMinutes(5),
    Priority = CacheItemPriority.High,
    SlidingExpiration = TimeSpan.FromMinutes(5),
};
memoryCache.Set(cacheKey, cachedData, cacheExpiryOptions);
```

Adding a sliding expiration to the cache

`MemoryCacheEntryOptions` provides four different types of configuration:

- SlidingExpiration – defines how long a cache entry can be inactive before it is removed from the cache
- AbsoluteExpiration – defines an exact date when an item is removed from the cache
- Priority – defines how important that cached object is. A higher priority object will be retrieved first

- Size – defines the size limit that entry

It is also possible to manually remove an item from the cache in code using `Remove()`:

```
public class HomeController : RenderController
{
    private readonly IMemoryCache _memoryCache;

    public HomeController(IMemoryCache memoryCache)
    {
        _memoryCache = memoryCache;
    }

    public override IActionResult Index()
    {
        var cacheKey = "myKey";

        _memoryCache.Remove(cacheKey);
    }
}
```

Removing an item from the cache

After combining these four types of caching strategies, you should see a noticeable improvement in your application's performance. After you have the basics, it is then a case of testing your page speed and tweaking until things are dialled in.

Further Reading

- Umbraco project structure and best practices
- WebEssentials.AspNetCore.OutputCaching
- More information on Tag helpers
- Caching in .NET Core
- Runtime bundling and Minification in ASP.NET Core with Smidge
- A guide to caching in ASP.NET Core
- Smidge
- Overview of porting from .NET Framework to .NET
- Ultimate Umbraco 9 Guide
- How To Deal With Content Migration In Umbraco 9

The End

Our current Umbraco journey together is now at an end. If you have followed the ideas and patterns in this book then you should have created a website that you can feel proud about. I want to thank you again for buying this book and for supporting me. It genuinely does mean a lot to me and it helps to keep me motivated moving forwards on my quest to help people make websites. So thank you and happy coding!

I welcome all feedback as feedback helps to make this book better value for everyone. If you want to send me any feedback then I would love to hear it. Please email me at umbracomastery@jondjones.com.