Optimizely

Content Cloud 12

Development Fundamentals

Exercise Book

September 2023

Content Cloud 12 – Development Fundamentals – Exercise Book

Course title: *Optimizely Content Cloud 12 – Development Fundamentals* Course code: 170-3020
Course version: August 3, 2022 Product Update 414, August 2, 2022

EPiServer.Templates version 1.2: .NET 6.0, `EPiServer.CMS.Core` 12.5.0, `EPiServer.CMS.UI` 12.6.0
EPiServer.Templates version 1.1: .NET 5.0, `EPiServer.CMS.Core` 12.3.2, `EPiServer.CMS.UI` 12.3.2

Optimizely CMS 12: `EPiServer.CMS.Core` 12.8.0, `EPiServer.CMS.UI` 12.9.0
Optimizely TinyMCE integration: `EPiServer.CMS.TinyMce` 3.3.0
Optimizely Forms: `EPiServer.Forms` 5.2.0
Optimizely Search & Navigation CMS integration (formerly Find): `EPiServer.Find.Cms` 14.1.0
Optimizely Search CMS integration (Lucene.NET): `EPiServer.Search.Cms` 10.0.0
Optimizely A/B Testing: `EPiServer. Marketing.Testing` 3.1.0
Optimizely Language Manager: `EPiServer.Labs.LanguageManager` 5.1.0

https://world.optimizely.com/releases/

Optimizely - update 414

Included services & packages: Content Cloud 11.36.10, PIM 1.1.0

Aug 02, 2022

New features for Optimizely PIM. Bug fixes for Optimizely Content Cloud.

Course Material Disclaimer

Important Disclaimer

This course material ("Course Material") has been prepared by Optimizely AB, Optimizely Inc. and various other subsidiaries ("Optimizely") based on information available from them and third party sources. By retaining this Course Material, you ("the Recipient" or "You") acknowledge and represent to Optimizely that You have read, understood and accepted the terms of this Important Notice. If You do not accept these terms, You should immediately destroy or delete this Course Material. This Course Material does not purport to contain all the information that You or a third-party may require in any connection with any business with Optimizely. You shall not use or rely on contents of this Course Material, or any information provided in connection with it, as product or service advice. No representation or warranty is made by Optimizely or any of its advisers, agents or employees as to the accuracy, completeness or reasonableness of the information in this Course Material or provided in connection with it. No information contained in this Course Material or any other written or oral communication in connection with it is, or shall be relied upon as, a promise or representation and no representation or warranty is made as to the accuracy or attainability of any estimates, forecasts or projections set out in this Course Material. No liability will attach to Optimizely, with respect to any such information, estimates, forecasts or projections. All third-party trademarks used within the Course Material are acknowledged, and used for only reference purposes.

Optimizely does not accept responsibility or liability for any loss or damage suffered or incurred by You or any other person or entity however caused (including, without limitation, negligence) relating in any way to this Course Material including, without limitation, the information contained in or provided in connection with it, any errors or omissions from it however caused (including without limitation, where caused by third parties), lack of accuracy, completeness, currency or reliability or You, or any other person or entity, placing any reliance on this Course Material, its accuracy, completeness, currency or reliability. Any liability of Optimizely (including advisers, agents and employees) to You or to any other person or entity arising out of this Course Material including any corresponding provision of any territory legislation, or any applicable law is, to the maximum extent permitted by law, expressly disclaimed and excluded.

You agree not to reproduce, print, re-transmit, copy, distribute, publish or sell any of the contents of this Course Material without the prior written consent of Optimizely. Reproduction of part or all of the contents of this Course Material in any form is expressly prohibited and may not be recopied and/or shared with a third party. The permission to recopy by an individual does not allow for incorporation of material or any part of it in any work or publication, whether in hard copy, electronic, or any other form.

Table of Contents

Optimizely Content Cloud – Development Fundamentals – Exercise Book

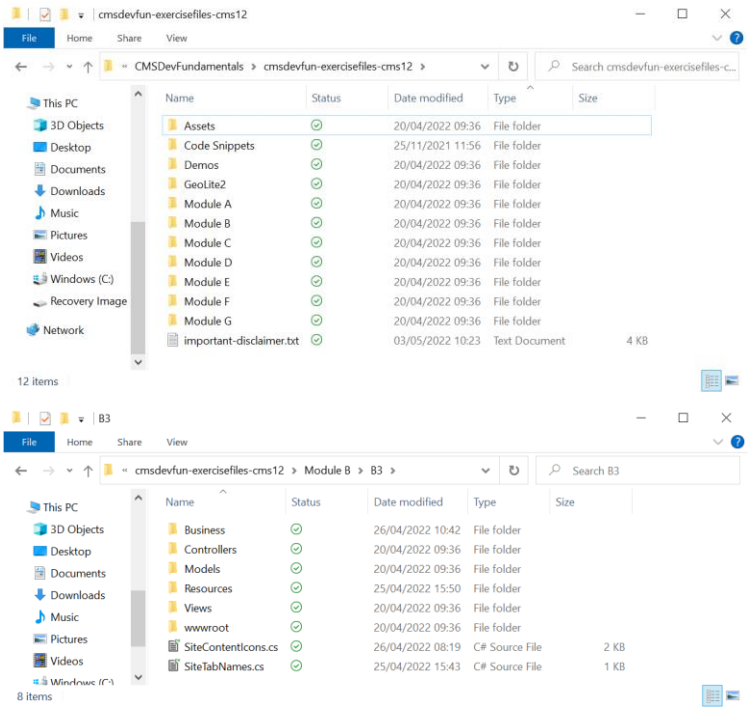
Student prerequisites

You should have experience using Microsoft Visual Studio to do ASP.NET MVC development.

Software requirements

To complete these exercises, we recommend:

- **Microsoft Windows 10** or later, or **Windows Server 2019** or later.
- **Microsoft Visual Studio 2022 for Windows** version 17.2 or later with latest updates.
  (The exercise book assumes you will use Visual Studio 2022 for Windows, so all screenshots and steps use this. You should also be able to use Visual Studio Code on Windows, macOS, or Linux, but some steps may be different so use at your own risk.)
- **EPiServer.Templates** version 1.6 or later and **EPiServer.Net.Cli** version 2.0 or later.
- **cmsdevfun-exercisefiles-cms12.zip** containing starter, solution, and support files for the exercises, as shown in the following screenshots:



Module A – Getting Started with Optimizely CMS

Goal

The overall goal of the exercises in this module is to learn how to use the core features of Optimizely CMS for editors and admins, and how to set up new websites. You will:

1. Set up an Alloy MVC sample website, update it to the latest version of CMS, and install some useful add-ons.
2. Review good practice for authentication and authorization, create common groups and users, and set access rights.
3. Practice creating, editing, and publishing content.
4. Define content approval sequences, approve content, and perform A/B testing.
5. Localize content for English, Swedish, and Danish languages, localize choices in the styles drop-down menu in the TinyMCE editor, and localize names and descriptions of content types.
6. Implement functionality to reset the administrator account in case of a forgotten password.
7. Download or clone and set up the Foundation - Frontline Services website starter project.
8. Identify common features on public websites that you might implement.

Exercise A1 – Setting up the AlloyDemo website

In this exercise, you will set up an Alloy MVC site, update it to use the latest version of CMS, and you will install an add-on that extends the CMS with editor-managed visitor forms capabilities.

- If you have already installed Visual Studio, set up the Optimizely NuGet package source, and installed the Optimizely project templates and CLI, then you can skip ahead to page 11.

Minimum development system requirements

https://docs.developers.optimizely.com/content-cloud/v12.0.0-content-cloud/docs/system-requirements-for-optimizely

- These instructions will use our most recent recommended development stack: Windows 10 or 11, Visual Studio 2022 including SQL Server Express LocalDb, and Optimizely project templates and CLI. Older versions may look and behave slightly differently. You do not need any specific coding tool, you could use Notepad if you want, but Visual Studio or Visual Studio Code have benefits like Intellisense and debugging.

Installing Microsoft Visual Studio Community 2022

- If you have already installed **Microsoft Visual Studio 2022** with the latest updates, then you can skip this section. .NET 6 development is not officially supported by Microsoft with Visual Studio 2019.

Version support from Microsoft:

- Visual Studio 2019 versions 16.0 to 16.8, and 16.10 are no longer under support.
- Visual Studio 2019 version 16.9 was a previous servicing baseline, and it will be supported with fixes and security updates until October 2022.
- Visual Studio 2019 version 16.11 is the current and last servicing baseline, and it will be supported with fixes and security updates until April 2029.
  https://devblogs.microsoft.com/visualstudio/visual-studio-16-11/
- Visual Studio 2022 is the current major version. It was released on November 8, 2021. **Episerver CMS Visual Studio Extension** does not support this version so you should continue to use Visual

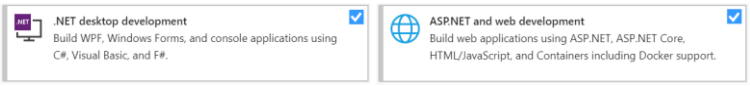
Studio 2019 if you need to create Optimizely CMS 11 projects. https://docs.microsoft.com/en-us/visualstudio/releases/2022/compatibility#-visual-studio-2022-support-for-net-development

- You can read about Visual Studio Product Lifecycle and Servicing at the following link: https://docs.microsoft.com/en-us/visualstudio/releases/2019/servicing

1. Download and install **Microsoft Visual Studio 2022** from the following link:

   https://visualstudio.microsoft.com/downloads/

2. At a minimum, choose the workloads: **.NET desktop development** and **ASP.NET and web development**, as shown in the following screenshots:



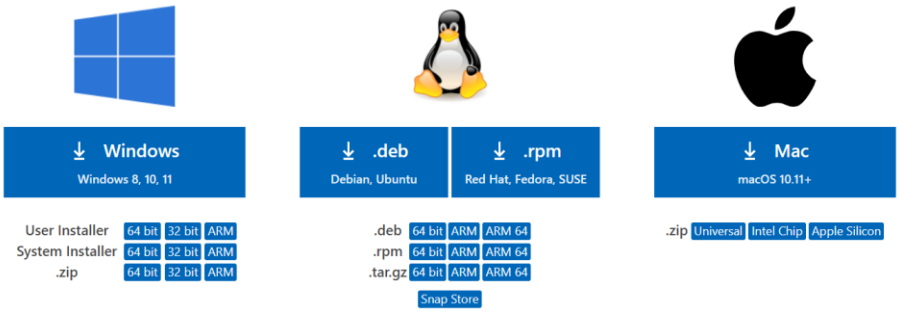
- Microsoft SQL Server Express LocalDb will be installed as part of **.NET desktop development** or you can install it separately from the following link: https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb

3. Optionally, add the workload: **Azure development**.

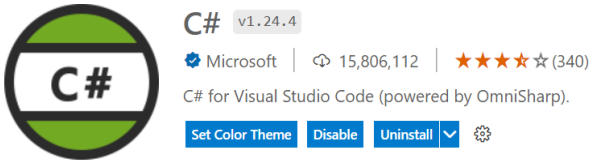
Installing Microsoft Visual Studio Code

- If you have already installed **Microsoft Visual Studio Code** with the latest updates, then you can skip this section.

1. Start your favorite browser.
2. Navigate to https://code.visualstudio.com/download
3. Click the appropriate download link for your operating system and CPU architecture and install Visual Studio Code, as shown in the following screenshot:



4. Start **Visual Studio Code**.
5. Navigate to **View** | **Extensions**, and search for the **C#** extension, and install it, as shown in the following screenshot:

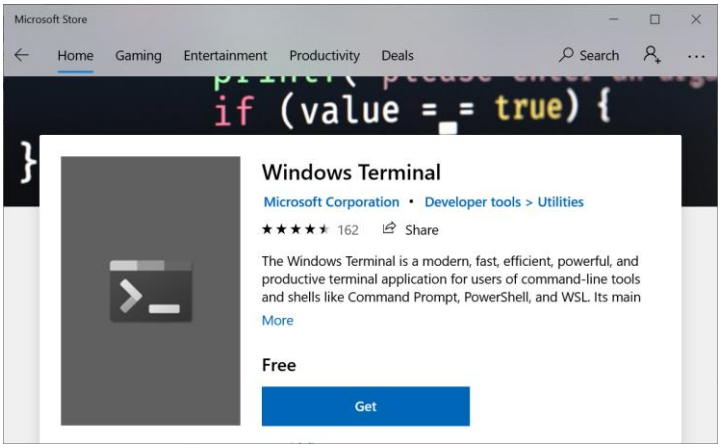


6. Search for and install the **SQL Server (mssql)** extension.
7. Search for and install the **MSBuild project tools** extension.
8. Search for and install the **REST Client** extension.
9. Search for and install the **ilspy-vscode** extension.

Installing Windows Terminal

A lot of modern .NET development happens at the command-line. The best tool is Windows Terminal, so we recommend that you install it:

1. Start **Microsoft Store**.
2. Search for **terminal** and note the Windows Terminal app, as shown in the following screenshot:



3. Click **Get**.

Installing .NET 5 or .NET 6 SDKs

- .NET 5 reached end-of-life on May 10, 2022. This does NOT mean it stops working! It just means after that date Microsoft will not provide technical support and will not release any bug fixes or security updates. We recommend using .NET 6.0 SDK which requires Optimizely project templates version 1.2 or later.

If you do not already have the latest .NET SDK 5.0 or .NET SDK 6.0 installed, then install them now:

1. Start command prompt or Windows Terminal.
2. List detailed information about .NET runtimes and SDKs, as shown in the following command:

```
dotnet --info
```

3. Note the typical partial results, as shown in the following output:

```
.NET SDK (reflecting any global.json):
 Version:   6.0.300
 Commit:    8473146e7d

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.19044
 OS Platform: Windows
 RID:         win10-x64
 Base Path:   C:\Program Files\dotnet\sdk\6.0.300\
```

```
Host (useful for support):
  Version: 6.0.5
  Commit:  70ae3df4a6

.NET SDKs installed:
  3.1.419 [C:\Program Files\dotnet\sdk]
  5.0.408 [C:\Program Files\dotnet\sdk]
  6.0.300 [C:\Program Files\dotnet\sdk]
...
```

- If you already have a .NET SDK 6.x.x installed, then you can skip to the next section, otherwise continue with the following steps to install the lastest .NET SDKs.

4. Start your favorite browser.
5. Navigate to https://dotnet.microsoft.com/download/dotnet/6.0.
6. In the **SDK** section, in the **Installers** column and **Windows** row, click **x64**, as shown in the screenshot:

- You must have Visual Studio 2022 version 17.2 or later to work with .NET SDK 6.0.

7. Run the installer e.g. `dotnet-sdk-6.0.300-win-x64.exe`.
8. Confirm that you now have .NET SDK 6.x.x by listing detailed information about .NET runtimes and SDKs, as shown in the following command:

   ```
   dotnet --info
   ```

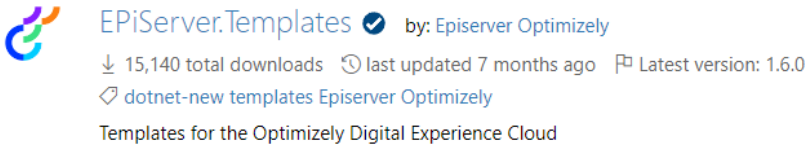
9. Close command prompt or Windows Terminal.

Installing the Optimizely project templates

- If you have already installed the **Optimizely project templates**, then you can skip this section.

We will start by ensuring you have installed the latest Optimizely `dotnet` project templates:

1. Start your favorite browser.
2. Navigate to https://www.nuget.org/profiles/Optimizely, and note that there is one package for templates but do not try to install it now from this web page, as shown in the following screenshot:



- I expect more Optimizely NuGet packages to be published to Microsoft's NuGet feed like this in future.

3. Start command prompt or Windows Terminal.

4. Install the Optimizely `dotnet` project templates, as shown in the following command:

```
dotnet new install EPiServer.Templates
```

- The `install` command will also upgrade the templates to a newer version if needed.

5. List the project templates that contain `optimizely`, as shown in the following command:

```
dotnet new list optimizely
```

6. Note the nine project and item templates, as shown in the following partial output:

```
These templates matched your input: 'optimizely'

Template Name                          Short Name                    Language    Tags
-------------------------------------  ----------------------------  ----------  --------------
Optimizely Alloy MVC                   epi-alloy-mvc                 [C#]        Web
Optimizely CMS Content Component       epi-cms-contentcomponent      [C#]        Web
Optimizely CMS Content Type Model      epi-cms-contenttype           [C#]        Web
Optimizely CMS empty                   epi-cms-empty                 [C#]        Web
Optimizely CMS Initialization Module   epi-cms-initializationmodule  [C#]        Web
Optimizely CMS Page Controller         epi-cms-pagecontroller        [C#]        Web
Optimizely CMS Razor Page              epi-cms-razorpage             [C#]        Web
Optimizely CMS Scheduled Job           epi-cms-job                   [C#]        Web
Optimizely Commerce empty              epi-commerce-empty            [C#]        Web
```

- There are three *project* templates (Optimizely Alloy MVC, Optimizely CMS empty, Optimizely Commerce empty) and six *item* templates (content component, content type, scheduled job, and so on). Optimizely Templates 1.1 was released on March 24th, 2022, and it creates projects that target .NET 5.0 and reference `EPiServer.CMS` version 12.3.2. Optimizely Templates 1.2 was released on May 11th, 2022, and it creates projects that target .NET 6.0 and reference `EPiServer.CMS` version 12.6.0. The latest version at the time of this writing is 1.6.0 and creates projects that target .NET 6.0 and EPiServer.CMS version 12.18.0.

Installing the Optimizely dotnet CLI database tool

- If you have already installed the **Optimizely dotnet CLI database tool**, then you can skip this section.

We will now ensure that you have installed the Optimizely `dotnet` CLI database tool:

1. Check if you have the Optimizely `dotnet` CLI tool already installed globally, as shown in the following command:

```
dotnet tool list --global
```

2. On my computer, I already have the tool installed, as shown in the following output:

```
Package Id              Version         Commands
---------------------------------------------------------
episerver.net.cli       2.0.0           dotnet-episerver
upgrade-assistant       0.3.310801      upgrade-assistant
```

3. Install (or update) the Optimizely `dotnet` command-line tool, as shown in the following command:

```
dotnet tool install EPiServer.Net.Cli --global --add-source
https://nuget.optimizely.com/feed/packages.svc/
```

- If the Optimizely `dotnet` CLI is already installed, then replace `install` with `update` and rerun the command to ensure that you have the latest version. At the time of writing in June 2023, the tool is published in the Optimizely NuGet feed. A newer version will probably be published to Microsoft's NuGet feed soon so you will not need to specify the `--add-source` switch and Optimizely NuGet feed URL.

Optional: Learning about the commands of the dotnet-episerver tool

1. Review the commands available when using the Optimizely `dotnet` CLI tool by entering the command `dotnet-episerver --help`, as shown in the following output:

- The hyphen between `dotnet` and `episerver` is optional, so `dotnet episerver --help` also works. Note that most switches like `--help` have a short version like `-h`. To help learning, I use the full switch name in this exercise book but feel free to try the short version if you prefer.

```
---------------------------------------
- Optimizely database cli tool, v1.0.0 -
---------------------------------------

Usage:
  dotnet-episerver [options] [command]

Options:
  --version         Show version information
  -?, -h, --help    Show help and usage information

Commands:
  create-cms-database <project>
  create-commerce-database <project>
  update-database <project>
  add-admin-user <project>
```

- Note this tool is used to create new databases and to update existing databases. The old `Update-EPiDatabase` command does not work for CMS 12 and Commerce 14 databases.

2. Review the options available when using the Optimizely `dotnet` CLI tool to create a new CMS database by entering the command `dotnet-episerver create-cms-database --help`, as shown in the following output:

```
---------------------------------------
- Optimizely database cli tool, v1.0.0 -
---------------------------------------

Usage:
  dotnet-episerver create-cms-database [options] <project>

Arguments:
  <project>

Options:
  -S, --server <server> (REQUIRED)                The server to create the cms database.
  -E, --integrated-security                       Use windows authentication when connecting
to server to create
                                                  database.
  -U, --username <username>                       The username when connecting to the server
to create the database.
  -P, --password <password>                       The password when connecting to the server
to create the database.
  -dn, --database-name <database-name>            The name of the database to create [default:
EPiServerDB_af0285cf]
  -du, --database-user <database-user>            The database user used for connectionstring.
[default:
                                                  EPiServerDB_af0285cfUser]
  -dp, --database-password <database-password>    The database password used for
connectionstring. [default:
                                                  A1FZupAD7izIjuwUnrkp$T3a2]
  -C, --collation <collation>                     The sql server collation [default:
SQL_Latin1_General_CP1_CI_AS]
  -v, --verbose                                   Enable verbose diagnostics
  -?, -h, --help                                  Show help and usage information
```

- With the older project templates, you would need to create the databases. With the new project templates the database is already created. But if you ever need to recreate a CMS database you now know how.

3. Discover how to create an admin user in the content database, as shown in the following command:

   ```
   dotnet-episerver add-admin-user --help
   ```

4. Note the help, as shown in the following output:

```
Usage:
  dotnet-episerver add-admin-user [options] <project>

Arguments:
  <project>

Options:
  -u, --username <username> (REQUIRED)              The username to create for the account.
  -e, --email <email> (REQUIRED)                    The email to create for the account.
  -p, --password <password> (REQUIRED)              The password to create for the account.
  -c, --connectionStringName <connectionStringName>  The connection string name with the
indentity core schema.  (REQUIRED)
  -v, --verbose                                     Enable verbose diagnostics
  -?, -h, --help                                    Show help and usage information
```

Configuring the Optimizely NuGet Feed package source

- If you have already configured the **Optimizely NuGet Feed** package source, then you can skip this section.

1. Start Microsoft Visual Studio 2022.
2. Navigate to **Tools** | **NuGet Package Manager** | **Package Manager Settings**.
3. In the **Options** dialog, in the list on the left, click **Package Sources**.
4. If the **Optimizely NuGet Feed** does not exist as an available package source, then click the **green plus** button to add it. The name can be anything, although we recommend using **Optimizely NuGet Feed**, and the path should be:

   `https://api.nuget.optimizely.com/v3/index.json`

- If you have other custom NuGet feeds configured and you get errors when installing or updating packages, try temporarily disabling all the extra feeds except Optimizely's and Microsoft's. If you have the old Optimizely or Episerver NuGet Feed, you can disable or remove it. The old Optimizely NuGet was: `https://nuget.optimizely.com/feed/packages.svc/`

Creating an Alloy MVC website project with sample content using command line

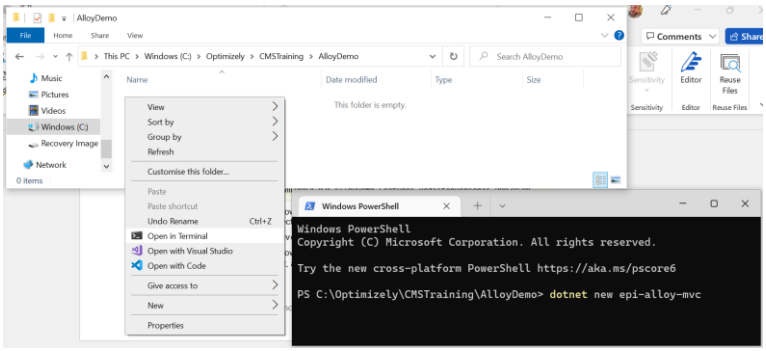
Now, we can create a new website project using the Alloy MVC project template and the `dotnet new` command. If you prefer to use Visual Studio 2022 then you can skip to the next section, but we recommend trying the command line first:

1. In a folder like `C:\Optimizely\CMSTraining`, create an empty folder for your project named `AlloyDemo`.

- You MUST name your project folder **AlloyDemo**. The solution classes have been written assuming that name and so all the namespaces will use it, e.g. `AlloyDemo.Features.RegisterPersonas`, and so on.

2. In command prompt or Windows Terminal, change to the **AlloyDemo** folder, and then create an Alloy MVC CMS website project, as shown in the following command and screenshot:

   ```
   dotnet new epi-alloy-mvc
   ```



3. In command prompt or Windows Terminal, enter the `dir` command and note the result is a new Alloy CMS 12 website project, as shown in the following output:

```
The template "Optimizely Alloy MVC" was created successfully.
PS C:\Optimizely\CMSTraining\AlloyDemo> dir

    Directory: C:\Optimizely\AlloyDemo

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        01/03/2022     16:19                App_Data
d-----        01/03/2022     16:16                Assets
d-----        01/03/2022     16:16                Business
d-----        01/03/2022     16:16                Components
d-----        01/03/2022     16:16                Controllers
d-----        01/03/2022     16:16                Extensions
d-----        01/03/2022     16:16                Helpers
d-----        01/03/2022     16:16                Models
d-----        01/03/2022     16:16                Properties
d-----        01/03/2022     16:16                Resources
d-----        01/03/2022     16:16                Views
d-----        01/03/2022     16:16                wwwroot
-a----        01/03/2022     16:16            242 .gitignore
-a----        01/03/2022     16:16            444 AlloyDemo.csproj
-a----        01/03/2022     16:16            694 appsettings.Development.json
-a----        01/03/2022     16:16            217 appsettings.json
-a----        01/03/2022     16:16            201 compilerconfig.json
-a----        01/03/2022     16:16           1321 compilerconfig.json.defaults
-a----        01/03/2022     16:16           2235 Globals.cs
-a----        01/03/2022     16:16            289 nuget.config
-a----        01/03/2022     16:16            481 Program.cs
-a----        01/03/2022     16:16            960 README.md
-a----        01/03/2022     16:16           1791 Startup.cs
```

4. In command prompt or Windows Terminal, enter a command to see the other switches when creating an Alloy MVC project, as shown in the following command:

   ```
   dotnet new epi-alloy-mvc --help
   ```

5. Note there are switches to set the SQL Server `sa` account password and to enable Docker support, as shown in the following output:

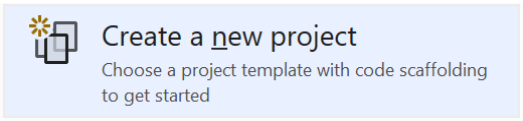
```
Optimizely Alloy MVC (C#)
Author: Episerver AB
Description: Example MVC application for testing and learning Optimizely CMS.
```

```
Options:
  --enable-docker  Enable Docker support
                   bool - Optional
                   Default: false

  --sa-password    The password the SA database account should have
                   string - Optional
                   Default: Qwerty12345!
```

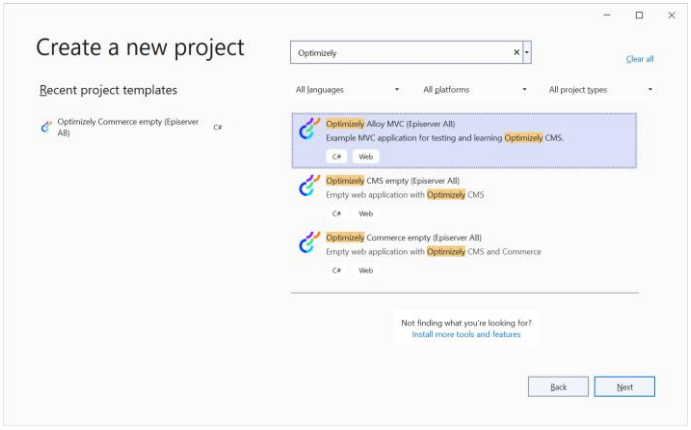
Optional: Creating an Alloy MVC website using Visual Studio 2022

Microsoft Visual Studio 2022 recognizes `dotnet new` project templates, so you can use it as an alternative to `dotnet new` at the command prompt. If you skipped creating the project using the `dotnet new` command, then you can follow these steps to use Visual Studio 2022:

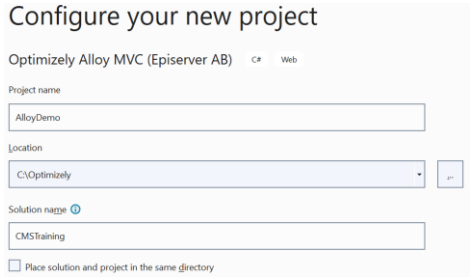
1. Start Microsoft Visual Studio 2022.
2. In the Start dialog, click **Create a new project**, as shown in the following screenshot:



3. In the **Search for templates** box, enter **Optimizely**, select **Optimizely Alloy MVC (Episerver AB)**, and then click **Next**, as shown in the following screenshot:



4. Enter the following options to configure your new project:



   - Project name: **AlloyDemo**

> ● You MUST name your project **AlloyDemo**. The solution classes have been written assuming that project name and so all the namespaces will use it, e.g. `AlloyDemo.Features.RegisterPersonas`, and so on.

   - Location: **C:\Optimizely** (or some other folder).
   - Solution name: **CMSTraining** (or something else unique).

- Place solution and project in the same directory: **No**

5. Click **Next**.
6. In **Additional information**, leave the defaults and then click **Create**, as shown in the following screenshot:

7. Wait for the project to be created.

Reviewing the Alloy MVC project

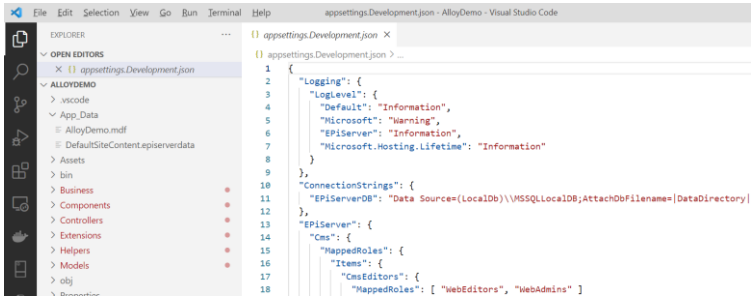
Now you can open the project in either Visual Studio 2022 or Visual Studio Code and review it:

1. Either open the **.csproj** file using Visual Studio 2022 or open the project folder using Visual Studio Code, as shown in the following command:

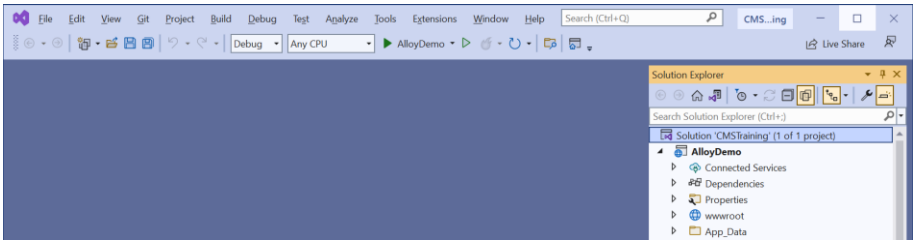
```
code .
```

- Entering `code .` means start Visual Studio Code and open the current folder. If you do not type the `.` it would only start Visual Studio Code.

2. Note the CMS database is created as expected in `App_Data` and the connection string is available in development mode in the `appsettings.Development.json` file, as shown in the following screenshot:



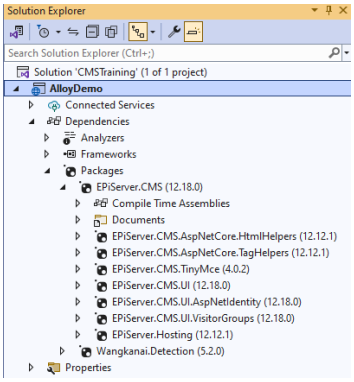
3. If you opened the project using Visual Studio 2022, then navigate to **File** | **Save Solution As...**, rename the solution file **CMSTraining.sln**, and save it in the **CMSTraining** folder.



4. In the project file `AlloyDemo.csproj`, note that it targets .NET 6 and CMS 12.18, and it uses .NET 6 features like implicit usings to import common namespaces globally including four commonly used `EPiServer` namespaces, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>disable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <Using Include="EPiServer"/>
    <Using Include="EPiServer.Core"/>
    <Using Include="EPiServer.DataAbstraction"/>
    <Using Include="EPiServer.DataAnnotations"/>
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="EPiServer.CMS" Version="12.18.0" />
    <PackageReference Include="Wangkanai.Detection" Version="5.2.0" />
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resources\Translations\**\*" />
  </ItemGroup>
</Project>
```

5. If you are using Visual Studio 2022, in the **Dependencies** folder, expand the x to show the dependent packages that are included with Optimizely CMS, as shown in the following screenshot:



- Each dependent NuGet package can have its own dependent packages forming a tree of dependencies.

6. In the `Properties` folder, in `launchSettings.json`, note the application URL setting, as shown in the following configuration:

```
{
  "profiles": {
    "AlloyDemo": {
      "commandName": "Project",
```

```
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5000/",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Starting the Alloy MVC website and registering an administrator account

Now you can start the website:

1. Using Visual Studio 2022's **Debug** | **Start Without Debugging**, or in command prompt or Windows Terminal or Visual Studio Code's Terminal, start the website project, as shown in the following command:

   ```
   dotnet run
   ```

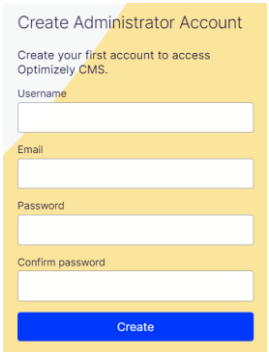
2. Note the output at the command line:

```
info: EPiServer.Framework.Initialization.InitializationEngine[0]
      Initialization started
info: EPiServer.Events.EventsInitialization[0]
      No event providers has been configured. Events will not be distributed to other servers.
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module CMS
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module Shell
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module EPiServer.Cms.UI.VisitorGroups
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module EPiServer.Cms.UI.Admin
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module EPiServer.Cms.TinyMce
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module EPiServer.Cms.UI.Settings
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module App
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolving Json serializer for Module 'EPiServer.Cms.UI.VisitorGroups' with setting
'ModuleJsonSerializerType=Net' and 'PreferredUiJsonSerializerType=Net'
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolving Json serializer for Module 'EPiServer.Cms.UI.Admin' with setting
'ModuleJsonSerializerType=None' and 'PreferredUiJsonSerializerType=Net'
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolving Json serializer for Module 'EPiServer.Cms.TinyMce' with setting
'ModuleJsonSerializerType=Resolve' and 'PreferredUiJsonSerializerType=Resolve'
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolved System.Text.Json serializer for Module 'EPiServer.Cms.TinyMce' according to global formatter
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolving Json serializer for Module 'EPiServer.Cms.UI.Settings' with setting
'ModuleJsonSerializerType=Net' and 'PreferredUiJsonSerializerType=Net'
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolving Json serializer for Module 'App' with setting 'ModuleJsonSerializerType=Resolve' and
'PreferredUiJsonSerializerType=Resolve'
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolved System.Text.Json serializer for Module 'App' according to global formatter
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolved System.Text.Json as serializer for CMS UI
info: EPiServer.Framework.Initialization.InitializationEngine[0]
      Initialization completed
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Optimizely\CMSTraining\AlloyDemo
```

3. In the output, *Ctrl* + click the URL for the website, https://localhost:5000/, as shown in the following screenshots:

4. If a browser does not start, then start your preferred browser manually and navigate to: `https://localhost:5000/`

5. Note that you are redirected to the Administrator register page, as shown in the screenshot:

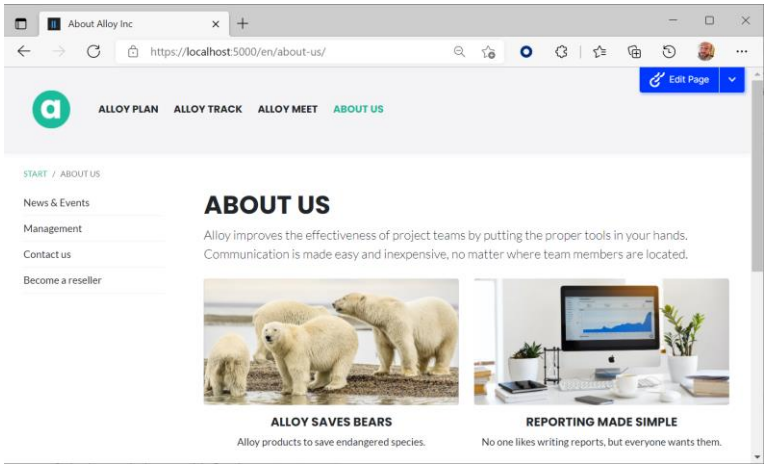


6. Enter suitable values:
   a. Username: `Admin`
   b. Email: admin@alloy.com
   c. Password: `Pa$$w0rd`

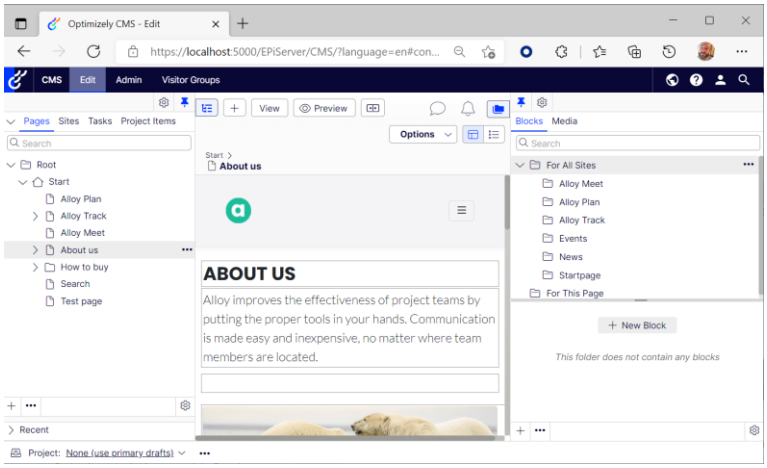
7. In command prompt or terminal, note the admin user registration page is enabled because (1) the browser is making a local request i.e. on the same computer as the web server, and (2) a single admin account does not yet exist, as shown in the following output:

```
info: EPiServer.Middleware.InitializeOnFirstRequestMiddleware[0]
      First request initialization completed for request
'https://localhost:5000/'
info: EPiServer.Cms.Shell.UI.Internal.RegisterAdminUserMiddleware[0]
      This is the first request to '/', redirecting to '/Util/Register'.
warn: EPiServer.Cms.Shell.UI.Controllers.Internal.RegisterAdminUserController[0]
      Access to the admin user registration is enabled. Following behaviors were evaluated 'Enabled,
LocalRequestsOnly, SingleUserOnly'.
warn: EPiServer.Cms.Shell.UI.Controllers.Internal.RegisterAdminUserController[0]
      Access to the admin user registration is enabled. Following behaviors were evaluated 'Enabled,
LocalRequestsOnly, SingleUserOnly'.
```

8. When the start page appears, note the Alloy website has navigation menu at the top, and click **About Us** to go to that page, as shown in the following screenshot:



9. In the quick access menu in the top right corner, click **Edit Page**, and note the **Edit** mode for the **About Us** page, and toggle on the left **Navigation** pane and the right **Assets** pane, as shown in the following screenshot:



- The CMS edit view is using an older menu system that runs along the top because the dotnet new template installed version 12.18 of **EPiServer.CMS,** in an upcoming step you'll upgrade to the latest version that has the main navigation along the left edge.

10. Close the browser.
11. At the command line or terminal, press *Ctrl + C* to shut down the Kestrel web server, and note the CMS initialization engine starts shutting down and completes the uninitialization process, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: EPiServer.Framework.Initialization.InitializationEngine[0]
      Uninitialization started
```

```
info: EPiServer.Framework.Initialization.InitializationEngine[0]
      Uninitialization completed
```

- In subsequent exercises, I will tell you to "Close the browser and shut down the web server." Remember that to do this, at the command line or terminal, press *Ctrl + C*. If you do not do this, you might see strange behavior the next time you start the website project.

Optional: Manually Updating the CMS database

You can check if an update to a NuGet package would require an update to the database schema at the following link: https://api.nuget.optimizely.com/databasecompare

- At the time of writing in May 2022 there are no database schema updates needed so you can skip this section if you like.

There are two ways to update the CMS database schema:

- Manually at the command line: `dotnet-episerver update-database ...`
- Automatically with an `appsettings.json` setting.

Recommended practice for deployments that you have control over e.g. on-premise, is to perform migrations manually, especially if you have written custom SQL scripts to manipulate the CMS database schema, for example, to improve DDS performance.

If you are deploying to Optimizely DXP cloud hosting, you must use the `appsettings.json` setting.

1. At the command prompt or Windows Terminal, update the CMS database, as shown in the following command:

   ```
   dotnet-episerver update-database AlloyDemo.csproj
   ```

12. Note that you get an error, as shown in the following output:

    ```
    [11:49:18 ERR] Update requires missing connection string EPiServerDB
    ```

13. In `appsettings.Development.json`, copy the setting for the connection string.
14. In `appsettings.json`, paste the setting for the connection string.
15. Modify the connection string to use an absolute path, as shown in the following configuration:

    ```
    "ConnectionStrings": {
      "EPiServerDB": "Data
    Source=(LocalDb)\\MSSQLLocalDB;AttachDbFilename=C:\\Optimizely\\CMSTraining\\AlloyDemo\\
    App_Data\\AlloyDemo.mdf;Initial Catalog=AlloyDemo;Integrated Security=True;Connect
    Timeout=30"
    }
    ```

16. At the command prompt or terminal, update the CMS database, as shown in the following command:

    ```
    dotnet-episerver update-database AlloyDemo.csproj
    ```

17. Note the successful messages, as shown in the following partial output:

    ```
    [11:55:50 INF] Running update command.
    [11:55:55 INF] Processing
    C:\Users\mapr\.nuget\packages\EPiServer.CMS.Core\12.3.0\tools\epiupdates\sql\7.8.0.sql
    ...
    [11:55:55 INF] Processing
    C:\Users\mapr\.nuget\packages\EPiServer.CMS.Core\12.3.0\tools\epiupdates\sql\12.2.1.sql
    [11:55:55 INF] Script validation: 0 - Already correct database version
    ```

Optional: Configuring automatic database updates

1. Open `appsettings.json`.
2. After the `AllowedHosts` setting, add a new setting to update the database schema automatically, as shown highlighted in the following configuration:

```
"AllowedHosts": "*",
"EPiServer": {
  "Cms": {
    "DataAccess": {
      "UpdateDatabaseSchema": "true"
    }
  }
}
```

- The various options settings are documented here: https://world.optimizely.com/documentation/developer-guides/CMS/configuration/Configuring-cms/#dataAccessOptions

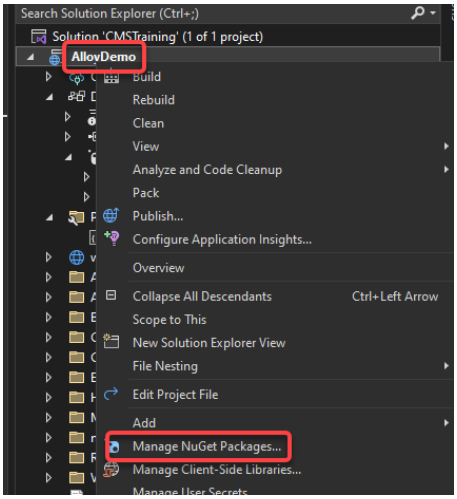
3. Save changes.

- When the site starts, it compares the assembly version and the database version. If the database version is lower than the assembly version and automatic updates are enabled, it applies the SQL schema updates. (The SQL files are embedded resources in the assembly.) You can read more about automatic schema updates at the following link: https://docs.developers.optimizely.com/content-cloud/v12.0.0-content-cloud/docs/configuring-cms#dataAccessOptions

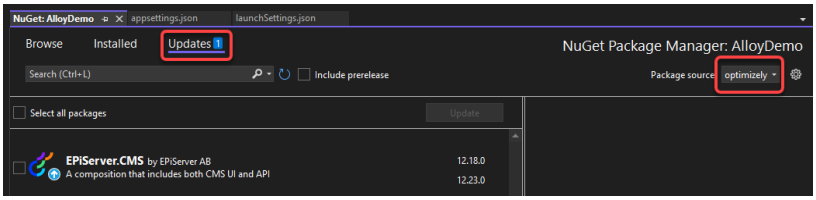
Updating to the latest CMS package

At the time of this writing, the **dotnet new** template installs version 12.18 of EPiServer.CMS. To get the latest navigation when editing and to match the screen captures going forward in the exercises, you'll need to update to at least version 12.22.

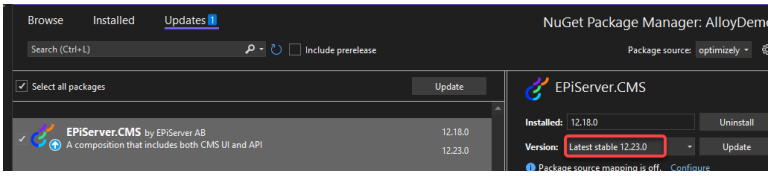
1. If you are using Visual Studio 2022, from the Solution Explorer panel, right click the **AlloyDemo** project and choose **Manage NuGet Packages** from the menu:



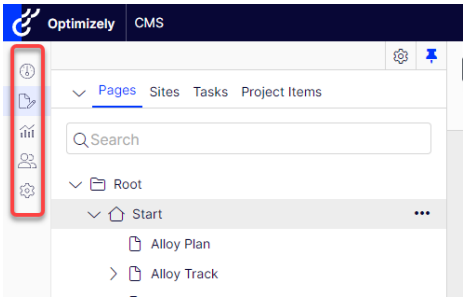
2. Change the **Package source** dropdown to the Optimizely source you configured in an earlier exercise, and switch to the **Updates** tab:



3. Select the **EPiServer.CMS** package and note the version number as well as dependencies:



4. Click the **Update** button and apply and accept the dependency and license dialogs.
5. Using Visual Studio 2022's **Debug** | **Start Without Debugging.**
6. Login as admin and switch to the editor view.
7. Note the CMS editor navigation is now icons along the left side of the UI:



Optional: Installing and configuring Optimizely Search & Navigation add-on

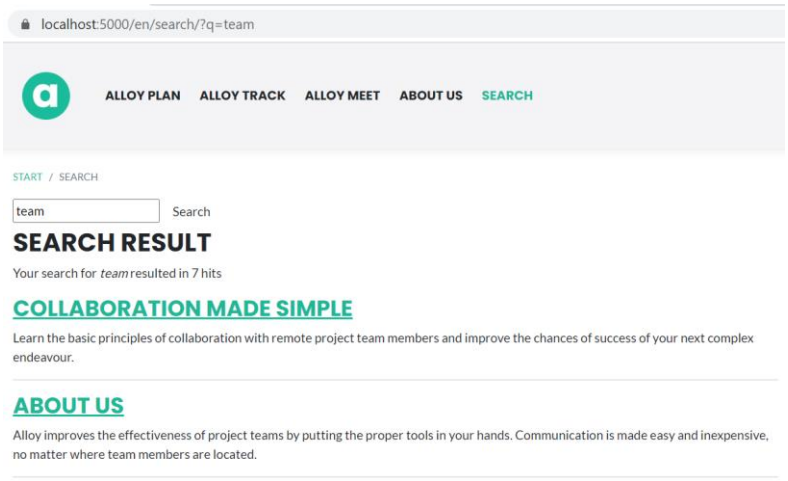
The Alloy project template has a page type and template for implementing a basic search page for visitors, but it is not enabled. We need to install and configure Optimizely Search & Navigation:

1. In **Exercise E4**, complete the tasks *Signing up for a free Search & Navigation index* on page 190 and *Installing and configuring Search & Navigation* on page 192.
2. In the **Controllers** folder, in **SearchPageController.cs**, modify the Index action method to implement unified search, as shown in the following code:

```
using AlloyDemo.Models.Pages;
using AlloyDemo.Models.ViewModels;
using EPiServer.Find; // IClient
using EPiServer.Find.UnifiedSearch; // UnifiedSearchResults
using Microsoft.AspNetCore.Mvc;

namespace AlloyDemo.Controllers
{
  public class SearchPageController : PageControllerBase<SearchPage>
  {
    private readonly IClient searchClient;
```

```
        public SearchPageController(IClient searchClient)
        {
          this.searchClient = searchClient;
        }

        public ViewResult Index(SearchPage currentPage, string q)
        {
          var model = new SearchContentModel(currentPage)
          {
            Hits = Enumerable.Empty<SearchContentModel.SearchHit>(),
            NumberOfHits = 0,
            SearchServiceDisabled = false,
            SearchedQuery = q
          };

          if (!string.IsNullOrWhiteSpace(q))
          {
            UnifiedSearchResults results = searchClient
              .UnifiedSearchFor(q).GetResult();

            model.Hits = results.Hits.Select(hit =>
              new SearchContentModel.SearchHit()
            {
              Title = hit.Document.Title,
              Url = hit.Document.Url,
              Excerpt = hit.Document.Excerpt
            });
            model.NumberOfHits = results.TotalMatching;
          }

          return View(model);
        }
      }
    }
```

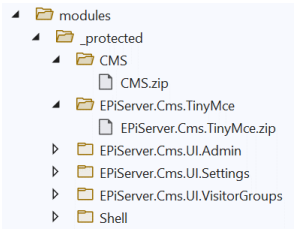
3. Start the project.
4. Log in as a CMS administrator.
5. In the **Pages** tree, select the **Search** page.
6. Select its **Display in navigation** check box.
7. Publish changes.
8. In the top menu, click the product selector and then click **Search & Navigation**.
9. Navigate to **Configure** | **Index**, and then click **Clear index**.
10. Click **Clear content index** and then click the popup link to navigate to the scheduled job.
11. Start the job manually and wait for it to finish.
12. Switch to Live view.
13. Click **Search**.
14. Enter **team**, click **Search**, and note the results, as shown in the following screenshot:



Installing Optimizely Forms add-on

Now we can install the Optimizely Forms add-on as a common example:

1. In the **AlloyDemo** project, expand the **modules_protected** folder, as shown in the screenshot:



- Optimizely products and add-ons are deployed as *shell modules*. In a new **Alloy MVC** project you will have seven modules already installed. The ZIP files contain the built-in functionality of the CMS, like Admin and Edit view. You can open the ZIP files and look at the contents, but don't change anything!

2. If you are using Visual Studio 2022, navigate to **Tools** | **NuGet Package Manager** | **Package Manager Console**.

- In **Package Manager Console**, you can press the up arrow ↑ to repeat a previous command. You can then edit the command before pressing *ENTER*. Package Manager Console NuGet commands and package names are not case-sensitive.

3. Enter the following command to install **Optimizely Forms**:

```
Install-Package EPiServer.Forms -ProjectName AlloyDemo
```

4. If you are using Visual Studio Code, in terminal, in the **AlloyDemo** folder, enter the following command to install **Optimizely Forms**:

```
dotnet add package EPiServer.Forms
```

5. In `AlloyDemo.csproj`, note the package reference for Optimizely Forms, as shown highlighted in the following markup:

```
<ItemGroup>
  <PackageReference Include="EPiServer.CMS" Version="12.23.0" />
  <PackageReference Include="EPiServer.Forms" Version="5.7.0" />
  <PackageReference Include="Wangkanai.Detection" Version="5.2.0" />
</ItemGroup>
```

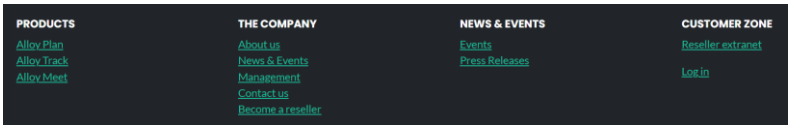
Exploring Optimizely Forms

Now we can see what extra functionality is added by the Optimizely Forms add-on:

1. Start the website project.
2. At the command prompt or Windows Terminal, note the info messages about Optimizely Forms, as well as the last info message indicating that because there is at least one admin account, the register page is disabled and will return a 404 if you attempt to navigate to it, as shown in the following partial output:

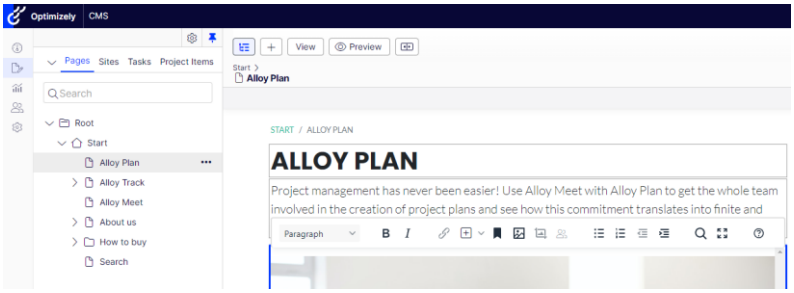
```
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module EPiServer.Forms.UI
...
info: EPiServer.Shell.Modules.ModuleTable[0]
      Adding module EPiServer.Forms
...
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolving Json serializer for Module 'EPiServer.Forms.UI' with setting
'ModuleJsonSerializerType=Resolve' and 'PreferredUiJsonSerializerType=Net'
info: EPiServer.Shell.Json.Internal.ShellModuleFormatterOptionsConfigurer[0]
      Resolved Newtonsoft serializer for Module 'EPiServer.Forms.UI' since it has custom
Newtonsoft converters
...
info: EPiServer.Forms.EditView.InitializationModule[0]
      Initialize EPiServer Forms.UI
...
info: EPiServer.Forms.EditView.Internal.FormUIInitializationService[0]
      Initialize EPiServer Forms: SetupRootFolderForForm
info: EPiServer.Forms.EditView.Internal.FormUIInitializationService[0]
      SetupRootFolderForForm: We need to create new folder as root for storing Form's Blocks
info: EPiServer.Forms.EditView.Internal.FormUIInitializationService[0]
      SetupRootFolderForForm: now Forms work with RootFolder for Block = 105
info: EPiServer.Forms.Helpers.Internal.FormsExtensions[0]
      Found 1 types, which implement (IsAssignableFrom)
EPiServer.Forms.Core.Feed.Internal.IFeedProvider
info: EPiServer.Forms.Helpers.Internal.FormsExtensions[0]
      Add to cache: EPiServer.Forms.Core.Feed.Internal.IFeedProvider, has 1 instance(s)
info: EPiServer.Forms.EditView.InitializationModule[0]
      Initialize EPiServer Forms: use custom FormViewEngines
...
info: EPiServer.Cms.Shell.UI.Internal.RegisterAdminUserMiddleware[0]
      The admin user registration is disabled, no redirect to '/Util/Register' will happen.
```

3. In the browser, scroll down the Start page, and in the **Customer Zone**, click **Log in**, as shown in the following screenshot:

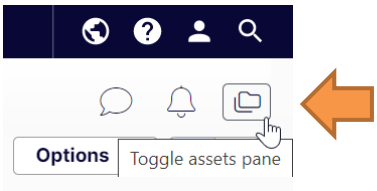


4. Log in as **Admin**.
5. In the quick access menu, click **Edit Page** to switch to **Edit** view for the start page.

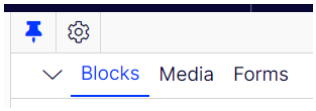
6. Navigate to one of the product pages, like **Alloy Plan**, click the **Main Body** property, and note the rich text editing toolbar including block style dropdown, bold, italic, links and image editing, bullets and numbering, find and replace, fullscreen mode and help, as shown in the following screenshot:



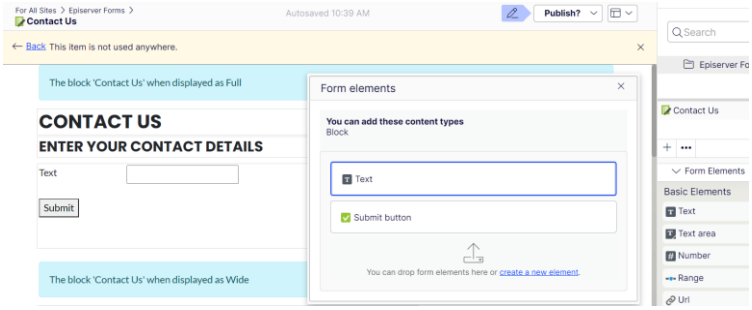
7. In **Edit view**, click the **Toggle assets pane** button, as shown in the following screenshot:



8. Click the **Pin** button to fix the assets pane to stay open.
9. In the **Assets** pane, note the **Forms** tab, next to **Blocks** and **Media**, as shown in the following screenshot:

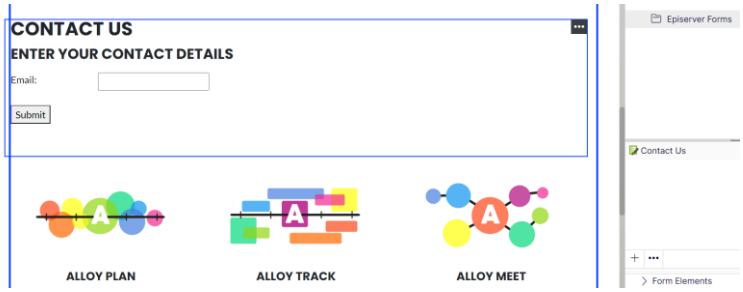


10. Click the **Forms** tab.
11. Click **+ New form**.
12. Enter **Contact Us** for the name.
13. Enter a form title and description, and drag and drop a **Text** element and a **Submit button** element into the **Form elements** area, as shown in the following screenshot:



14. On the context menu for the **Text** element, click **Edit**.
15. Set the **Name** to **Email**, the **Label text** to **Email:**, and in the **Validators** section, select the **Email** checkbox.

16. Click the **Publish?** Button and then click the **Publish Changes** button.
17. In the **Contact Us** form, click the **Publish?** Button and then click the **Publish** button.
18. Navigate to the Start page.
19. Drag and drop the **Contact Us** form into the content area above the three products, as shown in the following screenshot:



20. Publish changes to the Start page.
21. Switch to visitor view.
22. Enter an invalid email address and note the error message.
23. Enter a valid email address and note the success message.
24. Close the browser and shut down the web server (press *Ctrl + C* at the command line or terminal).

Optional: Customizing rich text editor using extension methods

Although initialization modules are still supported in CMS 12, we encourage the more modern use of extension methods called in the `Program` or `Startup` classes.

Let's see an example of how to replace an initialization module with modern configuration. In CMS 12, the `TinyMceInitialization` class has been removed, so we cannot create an initialization modile that has that as a dependency in cases where we want to customize the rich text editor. Instead, we must perform configuration in the `Startup` class.

1. In the **AlloyDemo** project, add a new class file named **TinyMceExtensions.cs**, and modify its contents, as shown in the following code:

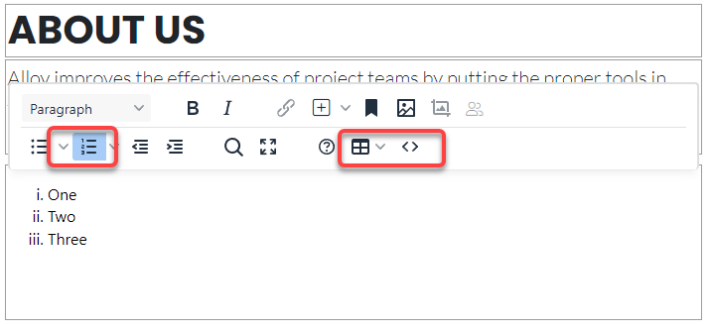
```
using EPiServer.Cms.TinyMce.Core; // TinyMceConfiguration
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace AlloyDemo
{
    public static class TinyMceExtensions
    {
        public static IServiceCollection AddTinyMceCustomizations(
            this IServiceCollection services)
        {
            services.Configure<TinyMceConfiguration>(config =>
            {
                // Add the advanced list styles, table, and code plugins
                // and append buttons for inserting and editing tables
                // and showing source code to the toolbar
                config.Default()
                    .AddPlugin("advlist")
                    .AddPlugin("table").AppendToolbar("table")
                    .AddPlugin("code").AppendToolbar("code");
```

```
            });

            return services;
        }
    }
}
```

2. In **Startup.cs**, at the bottom of the `ConfigureServices` method, add a statement to call the extension method, as shown highlighted in the following code:

   ```
   services.AddTinyMceCustomizations();
   ```

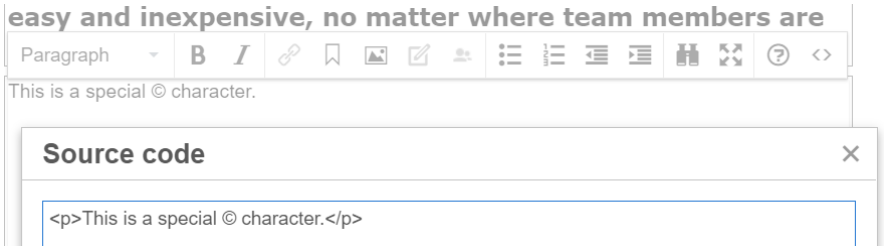
3. Save the changes.
4. Navigate to **Debug** | **Start Without Debugging** or press *Ctrl + F5*.
5. Log in as **Admin**, navigate to the **About us** page, and switch to **Edit** view.
6. Click in the **Main body** property and add some rich text, while noting the customized bullets, numbering, and additional buttons like tables and source code, as shown in the following screenshot:



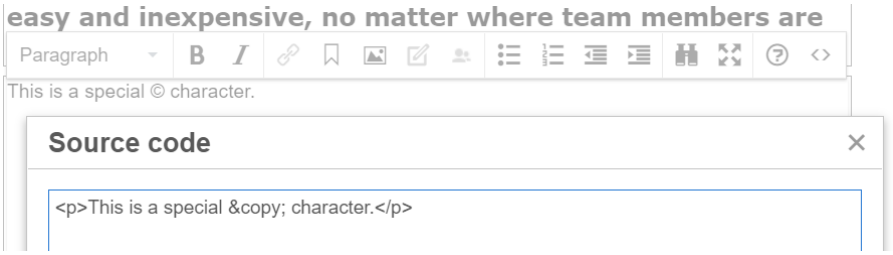
7. Close the browser and shut down the web server.

Optional: Controlling entity encoding in the source code dialog

Before CMS 11, viewing source code would show special characters like © as the character, as shown in the following screenshot:



With CMS 11, viewing source code shows special characters like © encoded by default, for example `&copy;` as shown in the following screenshot:



You can revert to the previous behavior by adding a setting, as shown in the following code:

```
config.Default().AddSetting("entity_encoding", "raw");
```

- Congratulations! You have now created a CMS website project, updated it to the latest version, and seen how to install a popular and useful add-on.

Exercise A2 – Reviewing and creating groups and users

In this exercise, you will create users and groups with different access rights on the website.

Prerequisites: complete Exercise A1.

Reviewing authentication and authorization in the Alloy MVC website

Optimizely Alloy MVC project template uses **ASP.NET Core Identity** with accounts stored in the CMS database to authenticate users and authorize roles and uses cookies for re-authentication.

1. In the **AlloyDemo** project, open **Startup.cs**. and review the `ConfigureServices` method, as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    if (_webHostingEnvironment.IsDevelopment())
    {
        AppDomain.CurrentDomain.SetData("DataDirectory", Path.Combine(
            _webHostingEnvironment.ContentRootPath, "App_Data"));

        services.Configure<SchedulerOptions>(
            options => options.Enabled = false);
    }

    services
        .AddCmsAspNetIdentity<ApplicationUser>()
        .AddCms()
        .AddAlloy()
        .AddAdminUserRegistration()
        .AddEmbeddedLocalization<Startup>();
}
```

Note:

- The call to `AddCmsAspNetIdentity` that integrates ASP.NET Core Identoty with the CMS for authentication purposes.
- Optimizely CMS has a custom extension method named **AddAdminUserRegistration** that enables the admin registration page. This page is only displayed to the visitor if, (1) the browser is executing on the web server i.e. it is a local request, and (2) there are no users in the CMS database. Now that we have registered an admin user, you could remove or comment out that statement.

2. Review the `Configure` method that sets up the HTTP request and response pipeline, as shown in the following code:

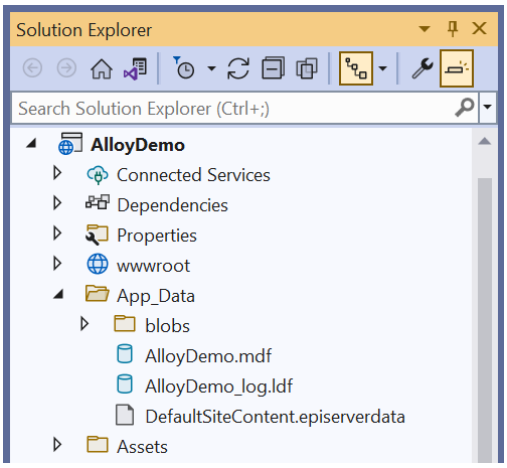
```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapContent();
```

```
        });
    }
```

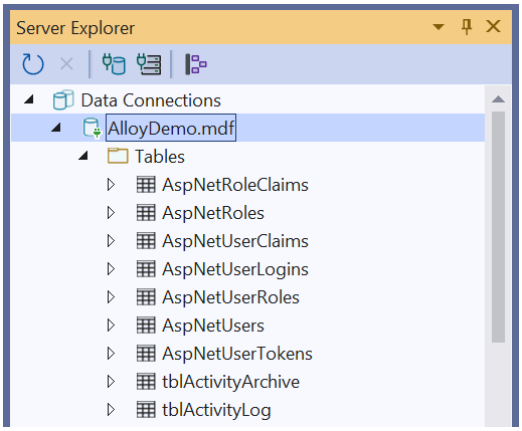
Note:

- The calls to `UseAuthentication` and `UseAuthorization` that perform security checks at that point in the pipeline.

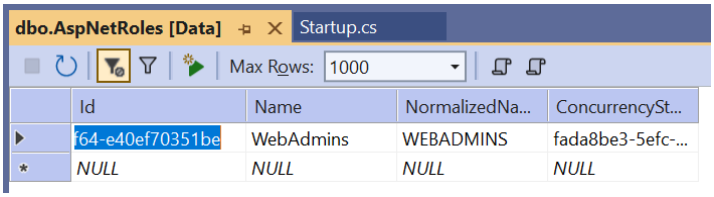
3. In **Solution Explorer**, expand the **AlloyDemo\App_Data** folder, and note the **blobs** folder, and the SQL database file, as shown in the following screenshot:



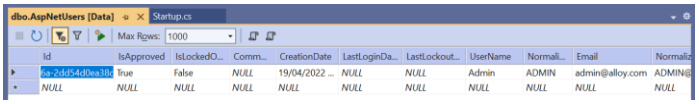
4. Double-click **AlloyDemo.mdf** to open a database connection to it.
5. In **Server Explorer**, expand **Tables**, and note the tables that begin with **AspNet**, as shown in the following screenshot:



6. Right-click **AspNetRoles**, and click **Show Table Data**, and note the **WebAdmins** role has been created for you, as shown in the following screenshot:



7. Right-click **AspNetUsers**, and click **Show Table Data**, and note the **Admin** user (or whatever you entered on the register page) has been created for you, and that it has columns for the following to implement best practice for security: **IsApproved**, **LastLoginDate**, **Email**, **PasswordHash** (the original password is not stored), **AccessFailedCount**, **UserName**, as shown in the following screenshot:



8. Close all the tables.
9. In **Server Explorer**, right-click the database, click **Close Connection**, and then close **Server Explorer**.
10. Close **Startup.cs**.

Reviewing virtual roles and changing the default mapping for CmsEditors

You will now follow good practice by reviewing and modifying the configuration of the virtual roles that give access to the working areas of CMS. Most projects should not leave the default configuration as-is so you will make some changes to learn some common modifications that you might make.

1. In the **AlloyDemo** project, open **appsettings.Development.json**.
2. Find the `MappedRoles` element, as shown in the following configuration:

```
"EPiServer": {
  "Cms": {
    "MappedRoles": {
      "Items": {
        "CmsEditors": {
          "MappedRoles": [ "WebEditors", "WebAdmins" ]
        },
        "CmsAdmins": {
          "MappedRoles": [ "WebAdmins" ]
        }
      }
    }
  }
}
```

- In this default configuration, users in the database-stored role named **WebAdmins** will become members of the virtual role named **CmsAdmins**, which gives access to all functionality within the CMS, but not necessarily content (but admins can always assign themselves access rights to content). Users in the database-stored role named **WebEditors**, will become members of the virtual role named **CmsEditors**, which gives access to **Edit** in the top menu. It is not necessary to map **WebAdmins** to **CmsEditors** so I think this is a mistake in the Alloy MVC project template.

3. Find the existing entry for **CmsAdmins**, and modify it so that a stored role named **AccessToAdminView** as well as **WebAdmins** maps to **CmsAdmins**, as shown in the following markup:

```
"CmsAdmins": {
  "MappedRoles": [ "WebAdmins", "AccessToAdminView" ]
```

```
    }
```

- You will create a stored role aka group named **AccessToAdminView** later in this exercise.

4. Find the existing entry for **CmsEditors**, and modify it so that a stored role named **AccessToEditView** maps to **CmsEditors** as well as **WebEditors**, as shown in the following markup:

```
"CmsEditors": {
  "MappedRoles": [ "WebEditors", "AccessToEditView" ]
},
```

- You will create a stored role aka group named **AccessToEditView** later in this exercise.

5. Add an entry to the list of virtual role providers, that maps members of a stored role named **Personalizers** to the virtual role named **VisitorGroupAdmins**, as shown in the following markup:

```
"VisitorGroupAdmins": {
  "MappedRoles": [ "Personalizers" ]
},
```

- You will create a stored role aka group named **Personalizers** later in this exercise. Any member of this group will have access to the **Visitor Groups** administration user interface.

6. Add an entry to the list of virtual role providers, that maps members of a stored role named **Developers** to the virtual role named **EPiBetaUsers**, as shown in the following markup:

```
"EPiBetaUsers": {
  "MappedRoles": [ "Developers" ]
},
```

- You will create a stored role aka group named **Developers** later in this exercise. Any member of this group will have access to any beta features of the user interface.

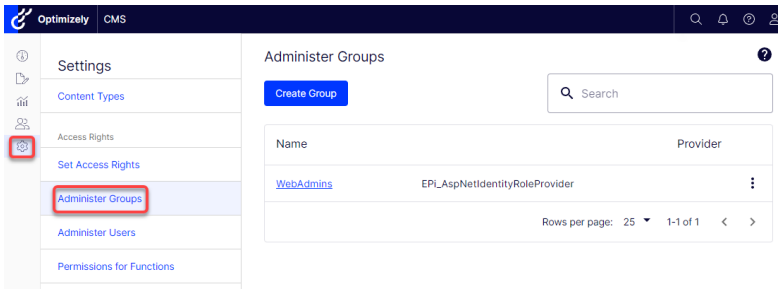
6. The complete `appsettings.Development.json` file should now look something like the following:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "EPiServer": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "ConnectionStrings": {
    "EPiServerDB": "Data
Source=(LocalDb)\\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\\AlloyDemo.mdf;Initial
Catalog=AlloyDemo;Integrated Security=True;Connect Timeout=30"
  },
  "EPiServer": {
    "Cms": {
      "MappedRoles": {
        "Items": {
          "VisitorGroupAdmins": {
            "MappedRoles": [ "Personalizers" ]
          },
          "EPiBetaUsers": {
```

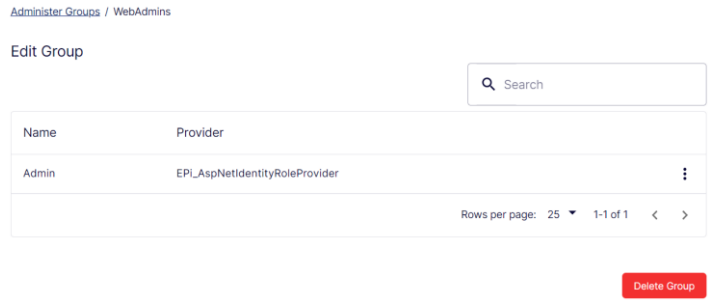
```
        "MappedRoles": [ "Developers" ]
      },
      "CmsEditors": {
        "MappedRoles": [ "WebEditors", "AccessToEditView" ]
      },
      "CmsAdmins": {
        "MappedRoles": [ "WebAdmins", "AccessToAdminView" ]
      }
    }
  }
 }
}
```

Reviewing the current groups and users

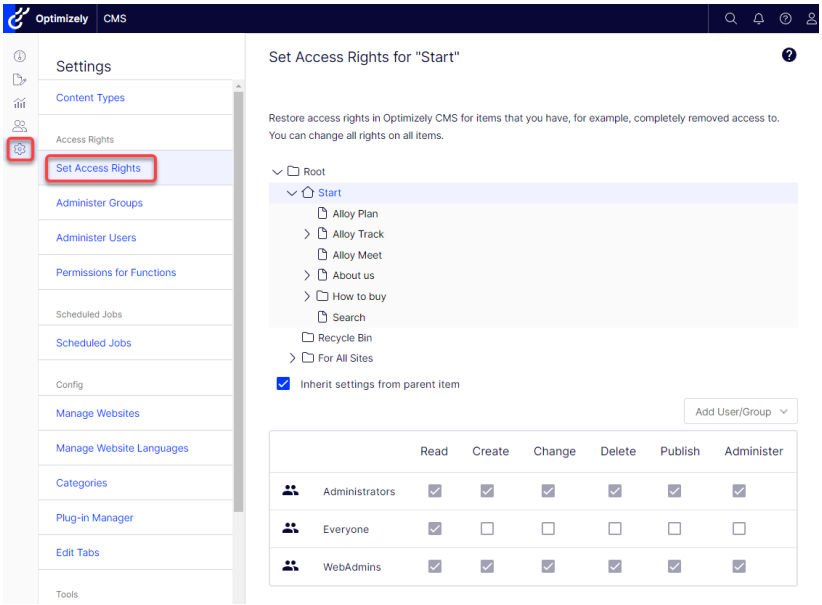
1. Start the **AlloyDemo** website, and log in as **Admin**.
2. Navigate to **Settings** | **Administer Groups**, as shown in the following screenshot:



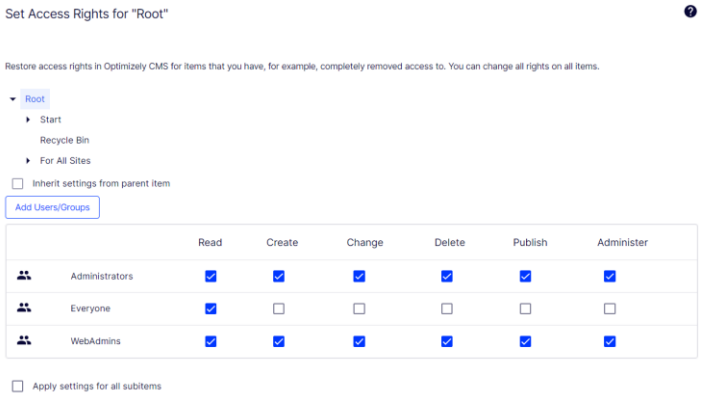
3. Click **WebAdmins**, to show the members of that group, as shown in the following screenshot:



4. Navigate to **Settings** | **Set Access Rights**, and note that by default the **Start** page (and all other pages) inherit their access rights from their parent item, as shown in the following screenshot:



5. Click **Root**, and note that it has entries for **Administrators** (usually the name of a Windows-stored group), **Everyone** (a virtual role) and **WebAdmins** (usually the name of an SQL-stored group in the CMS database), as shown in the following screenshot:



- **Everyone** is a special virtual role (it is not a mapped role) that automatically represents literally everyone including anonymous visitors, so by default they only have **Read** access rights to all content.

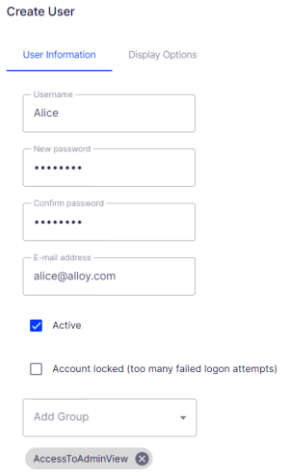
Creating some new groups and a user using Admin view

1. Navigate to **Settings** | **Administer Groups**.
2. Click **Create Group**.

3. Enter the name **AccessToEditView** and click **Create Group**.

- Remember that **AccessToEditView** is mapped to **CmsEditors** which gives access to **Edit** view.

4. Add a few more groups:
    a. **AccessToAdminView**: this is mapped to **CmsAdmins** which gives access to all working areas including Admin view.
    b. **ContentCreators**: members of this group will be able to create, change, and publish content throughout the site, except in the **News & Events** section.
    c. **NewsEditors**: members of this group will be the only editors who can create, change, and publish pages in the **News & Events** section.
    d. **Marketers**: members of this group will be the only users who can create **Product** pages.
    e. **Developers**: members of this group can access beta features.
5. Navigate to **Settings** | **Administer Users**, and then click **Create User**.
6. In the **Create User** page, fill in the following information, as shown in the following screenshot:
    a. Username: **Alice**
    b. Password: **Pa$$w0rd**
    c. E-mail address: **alice@alloy.com**
    d. Active: **selected**
    e. Member of: **AccessToAdminView**, **Marketers**, **NewsEditors**

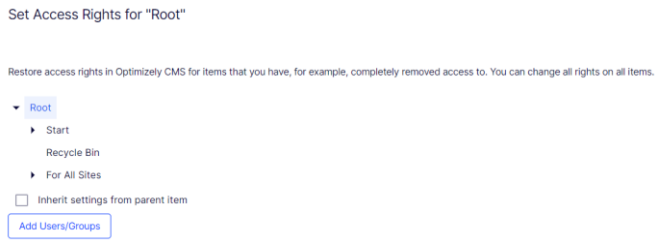


7. In the bottom right corner, click **Create User**.

Assigning default access rights to the Root page

1. Navigate to **Settings** | **Set Access Rights**.
2. In the content tree, select **Root**.

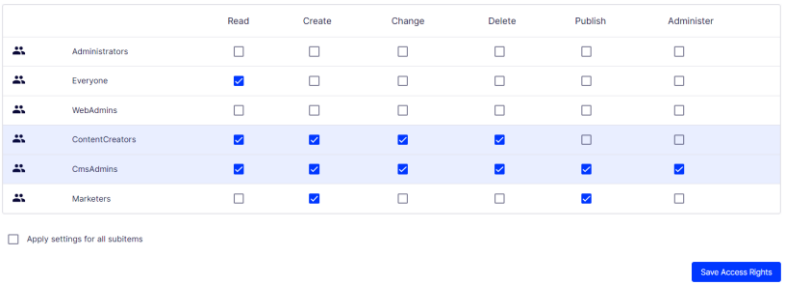
3. Click **Add Users/Group**, as shown in the following screenshot:



4. Click the **Groups** tab, and select **ContentCreators**, repeat for **Marketers**, and **CmsAdmins**.

- It is good practice to assign access rights to **CmsAdmins** instead of **WebAdmins** and **Administrators** because it uses the mapping in configuration and therefore provides flexibility.

5. Modify the following roles and their access rights, as shown in the following screenshot:



   a. **Administrators** and **WebAdmins**: clear all access rights.

- Clearing all access rights will remove the access control entry from the list when you click **Save Access Rights**.

   b. **CmsAdmins**: set **all** access rights.
   c. **ContentCreators**: set **Read**, **Create**, **Change**, **Delete** access rights.
   d. **Marketers**: set **Create** and **Publish** access rights.

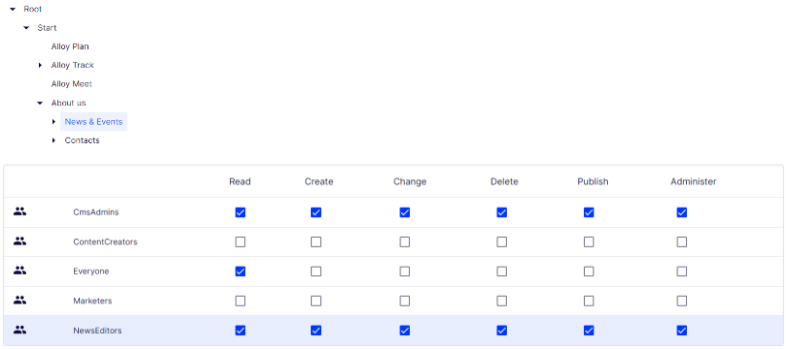
- Five of the six access rights (all except **Create**), apply to the content item that is selected, for example, **Root** in the previous screenshot. Applying **Create** access right to **Root** does not mean you can create the *root*. It means you can create children *underneath* **Root**.

6. Click **Save Access Rights**.

Assigning access rights to the News & Events page

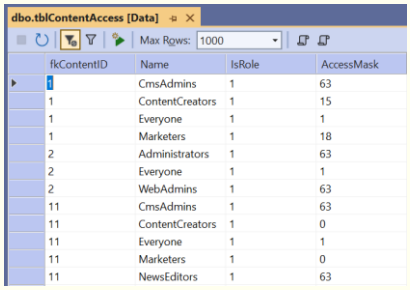
1. In the **Set Access Rights** content tree, expand **Start**, expand **About us**, and select **News & Events**.
2. Clear the **Inherit settings from parent item** check box.
3. Click **Add Users/Groups**.
4. Select **Groups**, add the **NewsEditors** group, and click **Add Users/Groups (1)**.

5. Clear all access rights from **ContentCreators** and **Marketers**, and assign all access rights to **NewsEditors**, as shown in the following screenshot:



6. Click **Save Access Rights**.

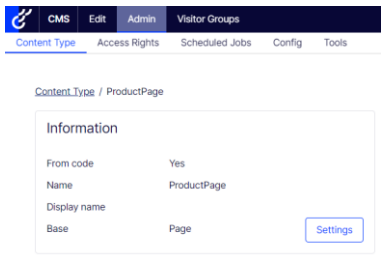
- If you were to open **AlloyDemo.mdf** and the table named **tblContentAccess**, you would see the following:



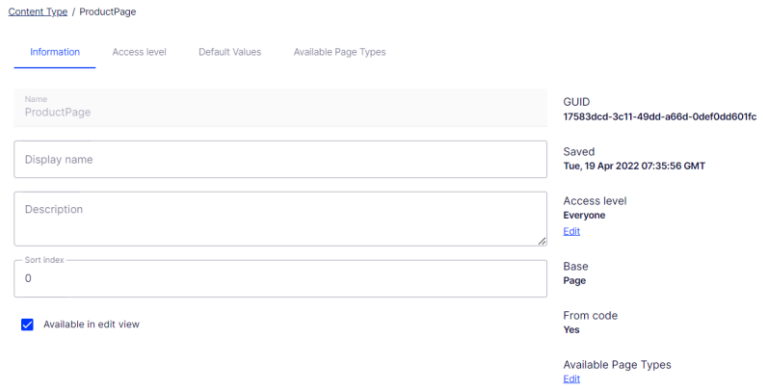
fkContentID	Name	IsRole	AccessMask
1	CmsAdmins	1	63
1	ContentCreators	1	15
1	Everyone	1	1
1	Marketers	1	18
2	Administrators	1	63
2	Everyone	1	1
2	WebAdmins	1	63
11	CmsAdmins	1	63
11	ContentCreators	1	0
11	Everyone	1	1
11	Marketers	1	0
11	NewsEditors	1	63

fkContentID: 1 = RootPage, 2 = Recycle Bin, 11 = News & Events
AccessMask: 0 = no access, 1 = read access, 63 = full access, and so on.

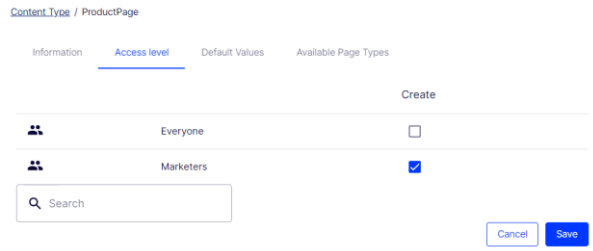
7. In **Admin** view, click **Content Type**, click **ProductPage**, and then click **Settings**, as shown in the following screenshot:



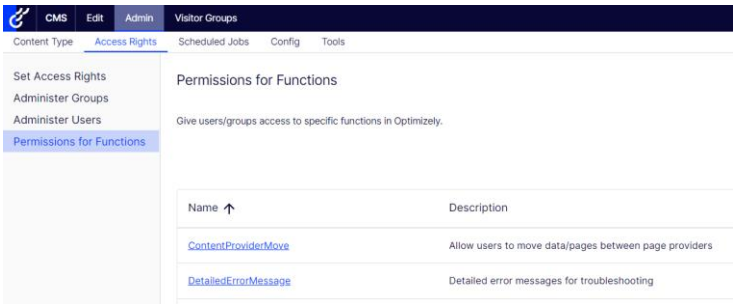
8. In the **Information** tab, in the **Access level** section, click **Edit**, or click the **Access level** tab, as shown in the following screenshot:



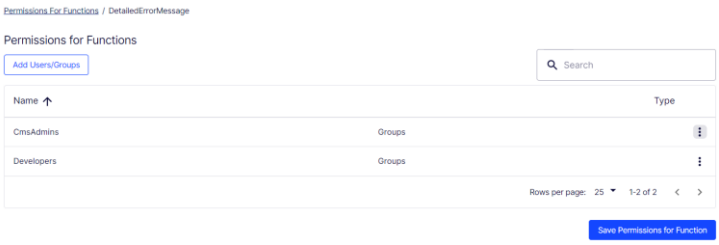
9. In the search box, enter the **Marketers** group and select it, and then clear the check box for **Everyone**, and click **Save**, as shown in the following screenshot:



10. Navigate to **Admin** | **Access Rights** | **Permissions for Functions**, and then click **DetailedErrorMessage**, as shown in the following screenshot:



11. Click **Add Users/Groups**, add the **CmsAdmins** and **Developers** groups, delete the **Administrators** group, and click **Save Permissions for Function**, as shown in the following screenshot:



12. Close the browser and shut down the web server.

Exercise A3 – Creating, editing, saving, and publishing content

In this exercise, you will get an understanding of how an editor works in the CMS. You will create a new page, add some links and images, and publish the page.

Prerequisites: complete Exercises A1 and A2.

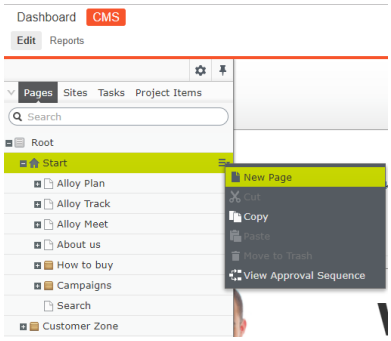
While working through these exercises, note the groups, users, access rights, and resources that were defined in Exercise A2, as summarized in the following table:

Groups	Users	Access rights	Resources
AccessToAdminView	Admin, Alice	All	All
AccessToAdminView, Developers	Dana	All	All
ContentCreators	Eve	Read, Create, Change, Delete	All content, except News & Events and Product pages
NewsEditors	Nick, Alice	Full	News & Events
Marketers	Michelle, Alice	Create, Publish	All content, except News & Events
		Create type	Product pages

Adding and publishing a new product page

1. Start the **AlloyDemo** website, log in as **Eve**, and note that she only has access to **CMS** | **Edit** and **CMS** | **Reports**, as shown in the following screenshot:

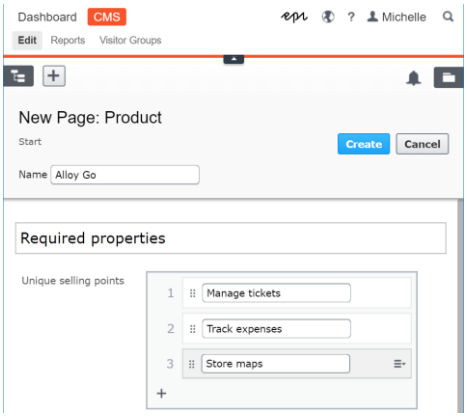
2. Under the **Start** page, add a new page, as shown in the screenshot:



3. In the list of page types, note that **Product** is not available for **Eve**, because only members of **Marketers** have access rights to create product pages.
4. Log out **Eve**, and log in as **Michelle**, and note that she has access to **CMS** | **Edit**, **CMS** | **Reports**, and **CMS** | **Visitor Groups**, as shown in the following screenshot:

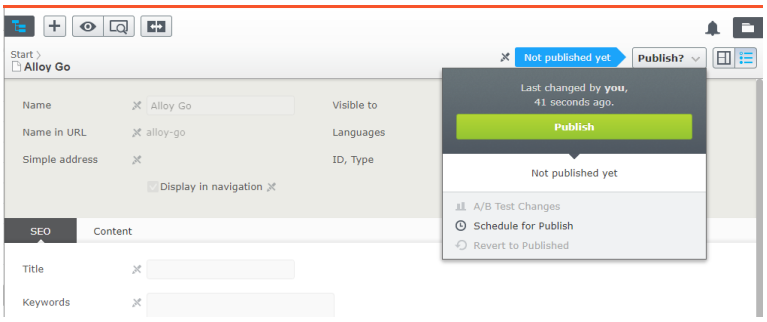
5. Add a new page under **Start**.

6. Click **Product**, enter the name **Alloy Go**, and enter some unique selling points, as shown in the following screenshot:



- Alloy Go will be a travel management software app that manages employee expenses, tickets, and maps.

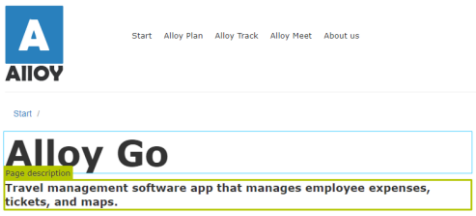
7. Click **Create**.
8. Switch to All Properties view and note that Michelle can create a product page, and she can publish it, but she cannot edit it, as shown in the following screenshot:



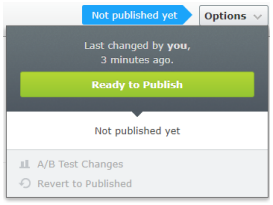
9. **Publish** the page.
10. Log out as **Michelle**.

Editing the product page

1. Log in as **Eve** and edit **Alloy Go**.
2. Enter a page description, as shown in the screenshot:
3. Enter some text for the main body. Be creative. Note that Eve has limited table customization options because she is not a member of **CmsAdmins**.
4. Click **Options** button and note that Eve cannot publish.



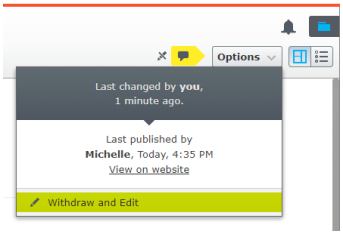
5. Click **Ready to Publish**, as shown in the screenshot:



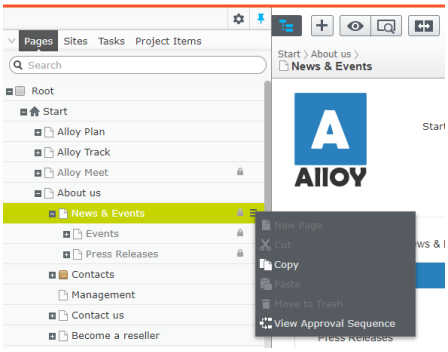
6. In **Assets** pane, select **Media** tab, and under the **For All Sites** folder, create a new folder named **Alloy Go**.
7. Use your favourite search engine to find some suitable images of travel related items and upload them to the **Alloy Go** folder.
8. Drag and drop the image(s) into the main body and explore the image editor feature.
9. Create a **Teaser** block in the **Alloy Go** folder.

- You will only be able to set the teaser to be **Ready to Publish** at this point.

10. Open the **Alloy Go** page. You previously set Alloy Go page to **Ready to Publish**, so it is locked. Click **Options**, and select **Withdraw and Edit**, as shown in the screenshot:



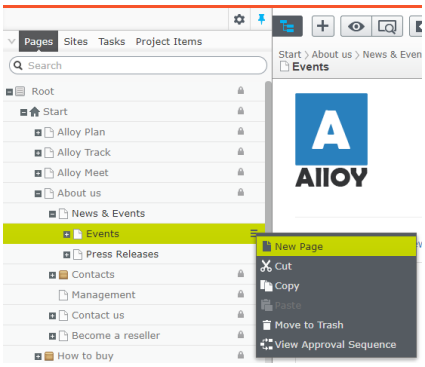
11. Drag and drop the teaser block you created into the **MainContentArea** on Alloy Go page.
12. Click the **Options** button and select **Ready to Publish**.
13. In **Navigation** pane, try to add a new page under **About us** | **News & Events**, and note that section and its children are locked for Eve, as shown in the following screenshot:



14. Log out as Eve.

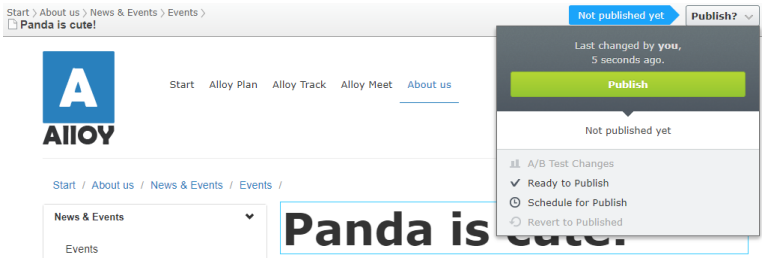
Adding and publishing a new article page

1. Log in as Nick.
2. In **Navigation** pane, try to add a new page under **About us** | **News & Events** | **Events**, and note that section and its children are available for Nick, but all other pages are locked, as shown in the screenshot:



3. Create an **Article** page under **About us** | **News & Events** | **Events**, named **Panda is cute!**

4. **Publish** the new page, as shown in the following screenshot:



5. Log out as **Nick**.
6. Log in as **Larry** and note that all content is locked for Larry.
7. Close the browser.

Exercise A4 – Personalizing and approving content

In this exercise, you will get an understanding of how personalization with visitor groups works, and how content approvals works. You will create a new page as one user, and then approve it as a sequence of other users.

Prerequisites: complete Exercises A1 and A2.

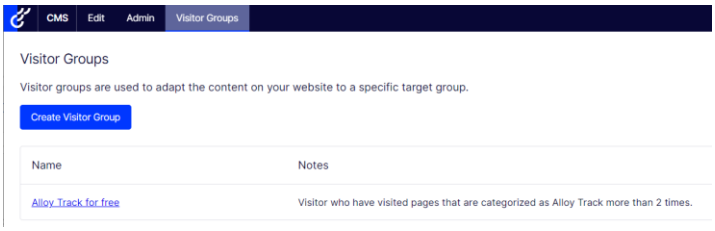
While working through these exercises, remember the groups, users, access rights, and resources that you defined in Exercise A2, as summarized in the following table:

Groups	Users	Access rights	Resources
ContentCreators	Eve	Full, except Administer	All content, except News & Events
NewsEditors	Nick	Full	News & Events
Marketers	Michelle	Create, Publish	All content, except News & Events
		Create type	Product pages
AccessToAdminView	Alice	All	All
Lawyers	Larry	Change (aka Edit)	Press Releases
CLevelExecs	Carlos	Create	Press Releases

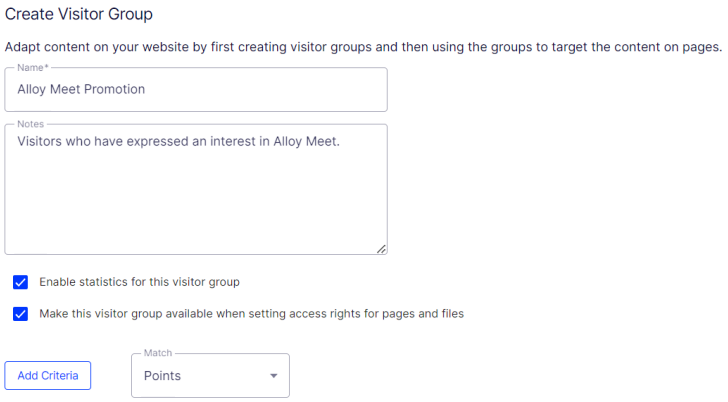
- We want Lawyers and CLevelExecs to be able to review Press Releases. There is no "review" access right. To enable them to review content, they must have Read and one other access right. We could give them Delete access right to enable them to review, but we don't want them to be able to delete press releases. The "minimum" access rights would therefore be Create because that only allows them to create children under the Press Releases page. If we deny them Create access rights to News pages, then they cannot.

Define some visitor groups for personalization

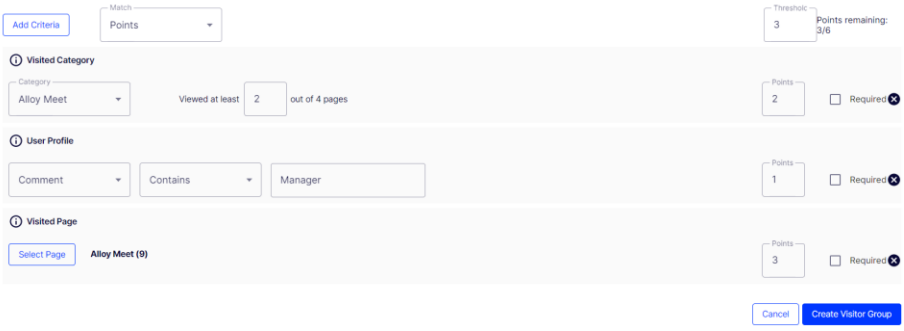
1. Start the **AlloyDemo** site and log in as **Alice**.
2. Navigate to **CMS** | **Visitor Groups** and click **Create Visitor Group**, as shown in the following screenshot:



3. In the **Create Visitor Group** page, enter the following as shown in the following screenshot:
   a. Name: **Alloy Meet Promotion**
   b. Notes: **Visitors who have expressed an interest in Alloy Meet.**
   c. Statistics: selected
   d. Security role: selected
   e. Match: **Points**



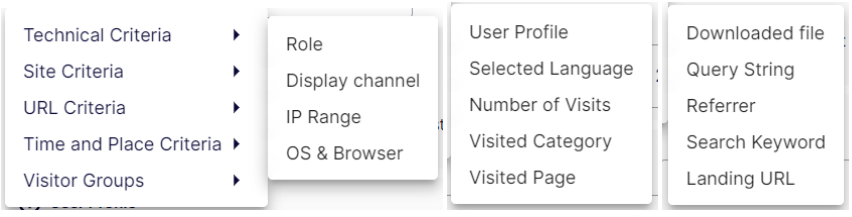
4. Click the **Add Criteria** button and select the following criteria, as shown in the following screenshot:
    a. Site Criteria > Visited Category: Alloy Meet, at least 2 pages; 2 points
    b. Site Criteria > User Profile: Comment contains Manager; 1 point
    c. Site Criteria > Visited Page: Alloy Meet; 3 points
    d. Site Criteria > Number of Visits: More than 3 within 60 days; 1 point

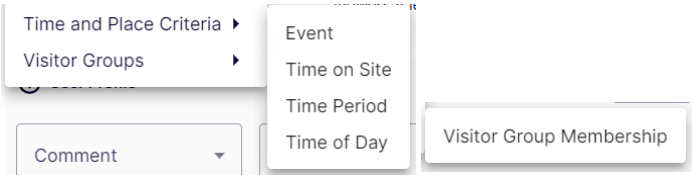


5. Set **Threshold** to 3 out of 6 points.
6. Click **Create Visitor Group**.
7. Close the browser and shut down the web server.

Install some extra visitor groups criteria

The default 19 visitor groups criteria are shown in the following screenshots:



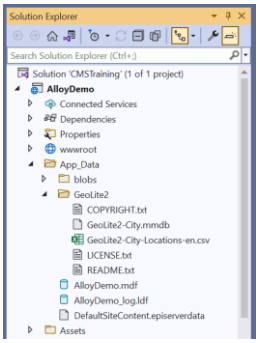


- You will only see **Geographic Coordinate** and **Geographic Location** criteria if you have configured a geolocation provider.

1. In Visual Studio, navigate to **Tools** | **NuGet Package Manager** | **Package Manager Console**.
2. Enter the following command:

   ```
   Install-Package EPiServer.Personalization.MaxMindGeolocation -ProjectName AlloyDemo
   ```

3. Drag and drop the **GeoLite2** folder from the solution ZIP to **App_Data**, as shown in the screenshot:



- Alternatively, navigate to https://dev.maxmind.com/geoip/geoip2/geolite2/ and download the MaxMind DB and Locations CSV files yourself to get the latest update. You can even create a scheduled job to download the latest update automatically using the instructions in the following article by Sanjay Kumar: https://world.optimizely.com/blogs/sanjay-kativar/dates/2021/5/update-geoip2-database-automatically/

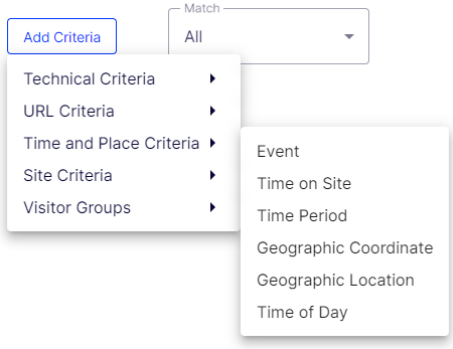
4. In **Startup.cs**, in the **ConfigureServices** method, add a call to add and configure the paths for the MaxMind database, as shown in the following code:

```
services.AddMaxMindGeolocationProvider(options =>
{
   options.DatabasePath = @"C:\Optimizely\CMSTraining\AlloyDemo\App_Data\GeoLite2\GeoLite2-City.mmdb";
   options.LocationsDatabasePath = @"C:\Optimizely\CMSTraining\AlloyDemo\App_Data\GeoLite2\GeoLite2-City-Locations-en.csv";
});

// alternative method
//services.AddMaxMindGeolocationProvider(
//    databasePath: @"C:\Optimizely\CMSTraining\AlloyDemo\App_Data\GeoLite2\GeoLite2-City.mmdb",
//    locationsDatabasePath: @"C:\Optimizely\CMSTraining\AlloyDemo\App_Data\GeoLite2\GeoLite2-City-Locations-en.csv");
```

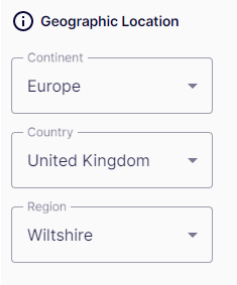
5. Start the website and log in as **Admin**.
6. Navigate to **CMS** | **Visitor Groups**.
7. Click **Create Visitor Group**.
8. At the command line or terminal, if you see the following error, then check the paths to the two files:

   ```
   fail: EPiServer.Personalization.MaxMindGeolocationProvider[0]
         Unable to find a geolocation database at the configured location 'C:\wrongfoldername\CMSTraining\AlloyDemo\App_Data\GeoLite2\GeoLite2-City.mmdb'. Some geolocation capabilities will be disabled.
   ```

9. Click **Add Criteria**, click **Time and Place Criteria**, and note the two geographic criteria, as shown in the following screenshot:

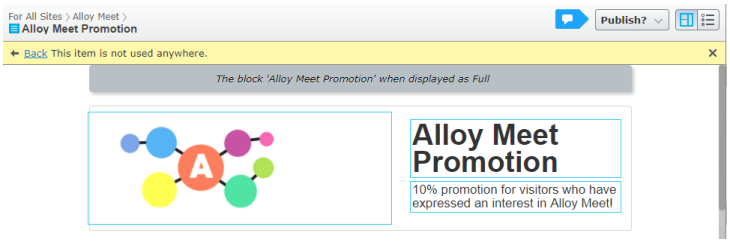


15. Select **Geographic Location**, and then select **Continent**, **Country**, and **Region**, for example, my home county, as shown in the following screenshot:



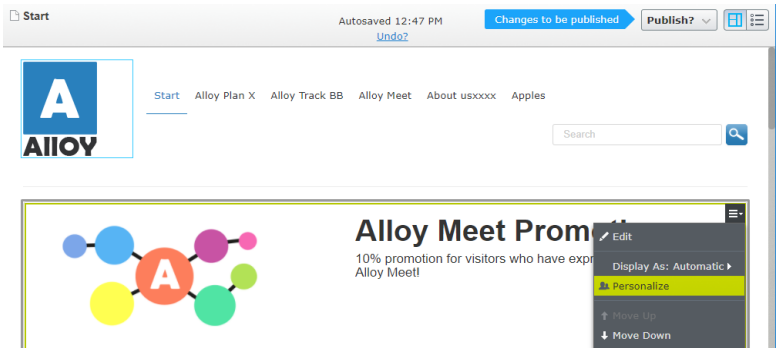
Create some personalized content

1. Start the website project and log in as **Admin**.
2. In **Edit** view, open **Assets** panes, on the **Blocks** tab, in the **Alloy Meet** folder, create a **Teaser** block named **Alloy Meet Promotion**, as shown in the following screenshot:

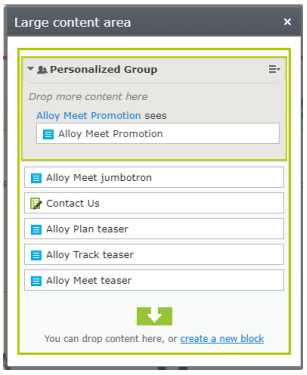


3. **Publish** the block.
4. Edit the **Start** page and drag and drop **Alloy Meet Promotion** to the top of its main content area.

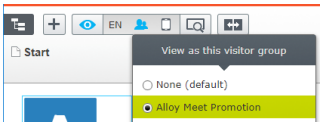
5. In the block's context menu, click **Personalize**, as shown in the following screenshot:



6. Select **Alloy Meet Promotion**, as shown in the following screenshot:



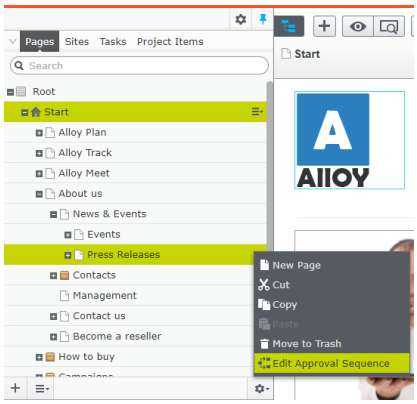
7. To preview the page as if you are a member of the visitor group, in the toolbar, toggle view settings, click **Alloy Meet Promotion**, as shown in the following sreeenshot:



Explore content approvals

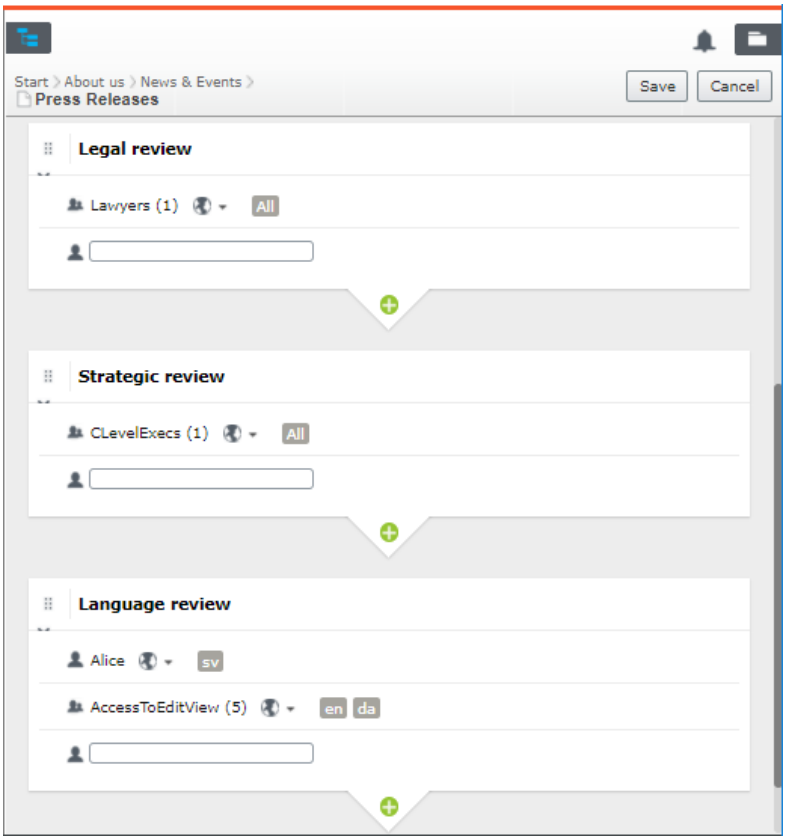
You can read the Optimizely CMS *Editor User Guide* to learn how to use this feature.

https://support.optimizely.com/hc/en-us/articles/4413192319757-Content-approvals



1. Open the **AlloyDemo** project.
2. Start the **AlloyDemo** site and log in as **Alice**.
3. Navigate to **CMS** | **Edit** and view the **Pages** in the **Navigation** pane.
4. Expand **About us** and **News & Events**.
5. Click the content menu for **Press Releases** and choose **Edit Approval Sequence**.

6. Set the approval sequence for the **Press Releases** page to **Enabled**.
7. Set **Require comment on Decline**.
8. Add three steps, and assign the groups that you created in Exercise A2, as shown in the following screenshot, and as listed in the following lettered bullets:
    a. Legal review by a lawyer
    b. Strategic review by a C-Level executive
    c. Review of Swedish content by Alice, who speaks Swedish fluently, and a review of other languages by anyone with access to Edit view



- It is good practice to use groups with at least two members as reviewers in each step. Assigning Alice to the Language review step is bad practice because she might not be available to approve or decline.

9. Save the sequence.

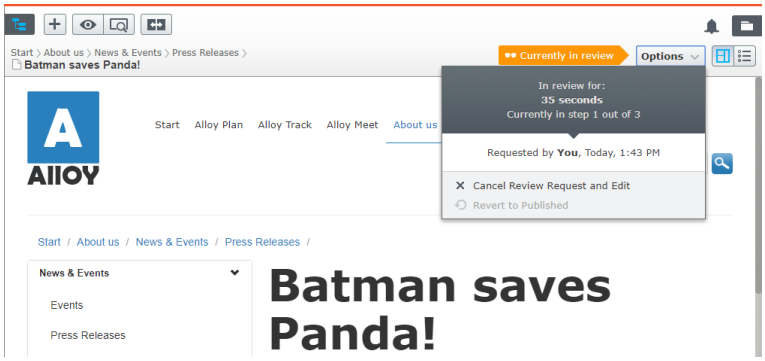
Enabling reviewers to approve or decline

1. In Admin view, navigate to **Admin** | **Access Rights** | **Set Access Rights**.
2. In the content tree, expand **About us**, expand **News & Events**, and select **Press Releases**.
3. Note the access rights for **Lawyers** and **CLevelExecs**, as shown in the screenshot:

- All users are members of **Everyone**, so all users have access rights to review content. To enable a user to approve or decline content they must have at least one other access right. We have given **CLevelExecs** the **Create** access right and **Lawyers** the **Change** access right. We could have given them **Delete** or **Publish** or **Administer** access rights too. Any of them enable a CMS user to approve or decline content that has an approval sequence.

Test the approval sequence

1. Log in as **Nick**.
2. Add a new **Article** to the **Press Releases** named **Batman saves Panda!**
3. Mark the page as **Ready for Review**, and note that the page is currently in review in step 1 of 3, as shown in the following screenshot:



4. Log out, and log in again as **Larry**.
5. Click the notification bell, click the notification, and **Approve** the article.
6. Log out, and log in again as **Carlos**.
7. Click the notification bell, click the notification, and **Approve** the article.
8. Log out, and log in again as **Nick**.
9. Click the notification bell, click the notification, and **Approve** the article.
10. Publish the article.
11. Repeat the process for a new article but see what happens when a lawyer declines approval.

Optional: Commonly Underutilized CMS Editor Features

"Through my years of working with Episerver CMS I've strived to take advantage of any features that increase efficiency. I'm surprised as I work with even seasoned CMS editors that some common features are often underutilized. These 5 features you're probably not using will help make your CMS editor experience more productive." – Glenn Lalas

https://www.adagetechnologies.com/5-episerver-cms-editor-features/

Exercise A5 – Localizing content

In this exercise, you will localize some content into Swedish and Danish, including pages and blocks.

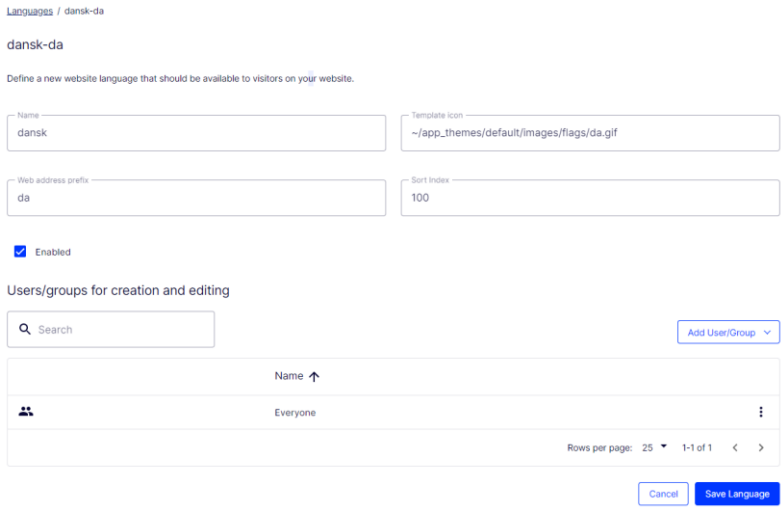
Prerequisites: complete Exercise A1.

Enabling Danish language for the website content

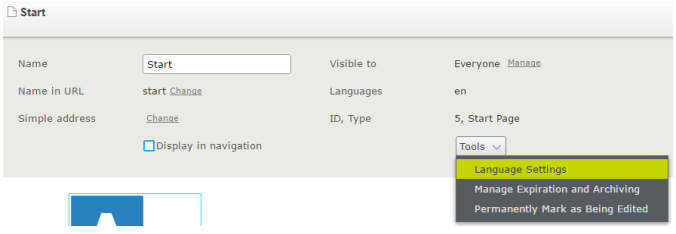
1. Open the solution with the **AlloyDemo** project.

- Make sure you use **AlloyDemo**, so you have lots of sample content pages that you can translate.

2. Start the site, and log in as **Admin**.
3. Navigate to **CMS** | **Admin** | **Config** | **Manage Website Languages**, and then note that English and Swedish are already enabled by the **Alloy MVC** project template.
4. Click **dansk** (Danish) language.
5. Check the **Enabled** box, and then click **Save Language**, as shown in the following screenshot:

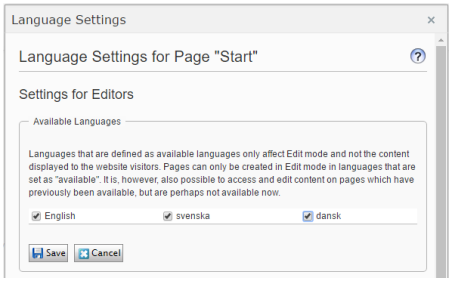


6. Navigate to **CMS** | **Edit** | **Pages** | **Start** page.
7. Click **Tools** | **Language Settings**, as shown in the following screenshot:



8. In **Settings for Editors**, click **Change**.

9. Check the **dansk** box, and click **Save**, as shown in the following screenshot:



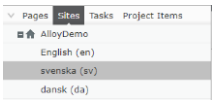
10. In **Settings for Site Visitors** | **Fallback Languages**, click **Change**.
11. Set **Swedish** to fallback to **Danish**, and then **English**.
12. Set **Danish** to fallback to **Swedish**, with no second fallback, then click **Save**, as shown in the following screenshot:

- We could have given Danish a second fall back, but in the following steps, you will see what happens if you don't have one.

13. Close the **Language Settings** dialog box.

Translating content

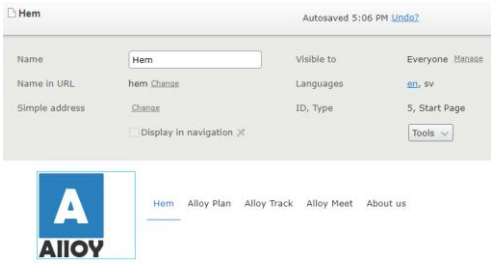
1. In the **Navigation** pane, click **Sites**, and then switch to the **svenska** (Swedish) site, as shown in the screenshot:



2. Note the **Start** page has already been "translated" into Swedish in the Alloy MVC project template.

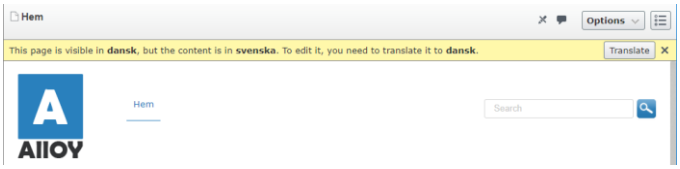
- The Swedish Start page doesn't have any actual content so its **Large content area** is empty!

3. Edit the name of the page to **Hem** (home in Swedish) and the **Name in URL** to **hem**, as shown in the following screenshot:



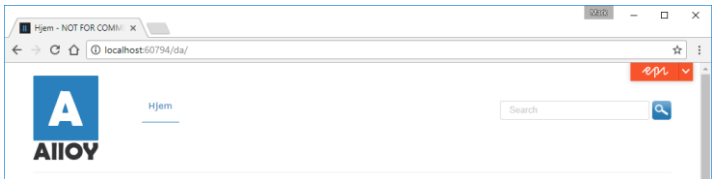
- You can quickly toggle between languages for a page by clicking the language code links in the basic information area.

4. Publish the changes.
5. In **Navigation**, click **Sites**, and switch to the Danish language site, and note the page is visible to a visitor who asks for Danish (dansk) because it falls back to Swedish, as shown in the following screenshot:

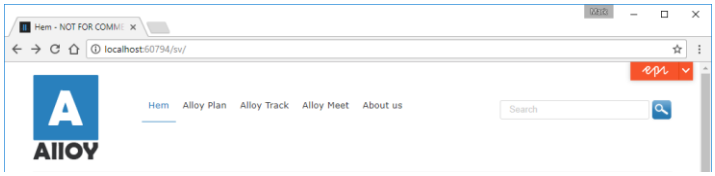


6. Click **Translate**. Note the page is NOT translated automatically for you. You must translate the text yourself (**Hjem** is Danish for home), and then click **Create**, as shown in the following screenshot:

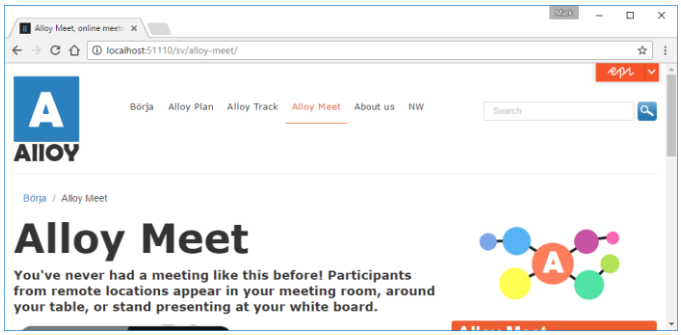
7. Publish the page.
8. View the site as a visitor and enter `/da/` at the end of the address box. Note that when viewed in Danish, the Danish Start page (which is empty) is displayed, and there are no links to other pages like products, because Danish can only fallback to Swedish pages, as shown in the following screenshot:



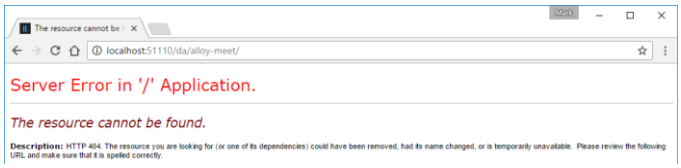
9. Change `/da/` to `/sv/` to request the Swedish start page, and note that due to fallback language settings, when content is not available in Swedish, it can fall back to English content, for example links to product pages, as shown in the following screenshot:



10. View the **Alloy Meet** page. It falls back to English, as shown in the following screenshot:



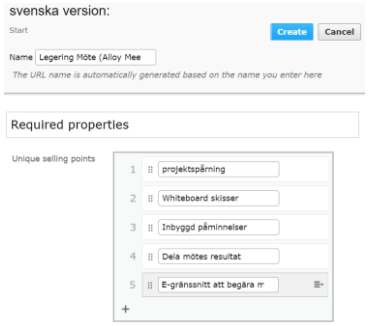
11. In the address bar, change `sv` to `da`, and note the 404 error, due to strict language routing rules, as shown in the following screenshot:



12. Switch to **Edit** view.

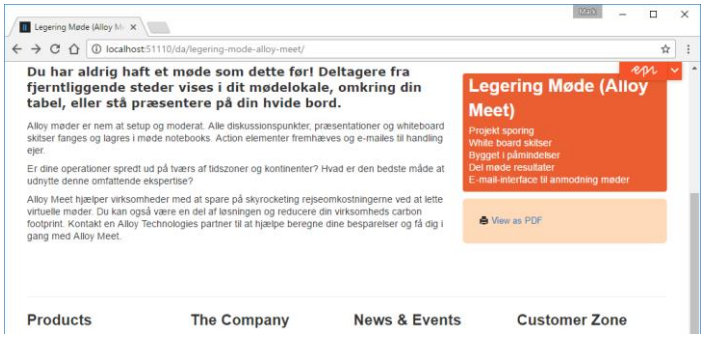
- There is a file in **\cmsdevfun-exercisefiles\Module A\A5**, named **translations.pdf**, with translations for some pages and blocks in English, Swedish, and Danish.

13. Translate the **Alloy Meet** page into Swedish, as shown in the following screenshot:

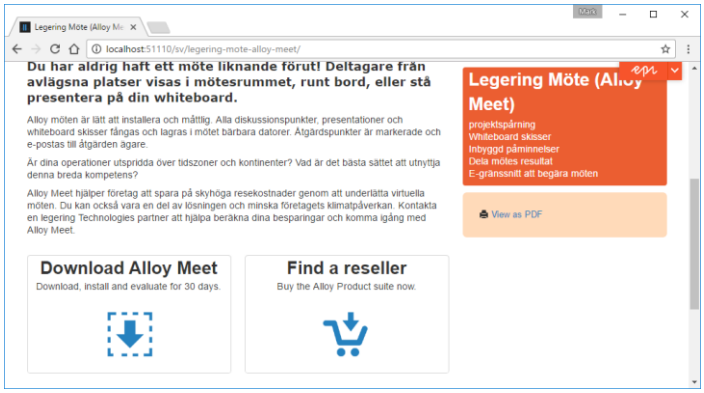


14. Translate the **Heading**, **Page description**, and **Main body** properties using the **translations.pdf** file, as shown in the following screenshot:

15. Publish the page.
16. Switch to the Danish site, translate and publish the Danish version of the Alloy Meet page.
17. View the site as a visitor. The Danish Alloy Meet page does not show blocks, because the blocks only exist in English, not Danish or Swedish, as shown in the following screenshot:



18. The Swedish version of the Alloy Meet page does show blocks (in English), as shown in the following screenshot:



Localizing content areas and blocks

`ProductPage` inherits from `StandardPage`, which has a `MainContentArea` that is not localized (because the property does not have `[CultureSpecific]` applied), so block references are shared by all language branches.

1. Open **~/Models/Pages/StandardPage.cs**, and note the `MainContentArea` property is not culture specific, as shown in the following code snippet:

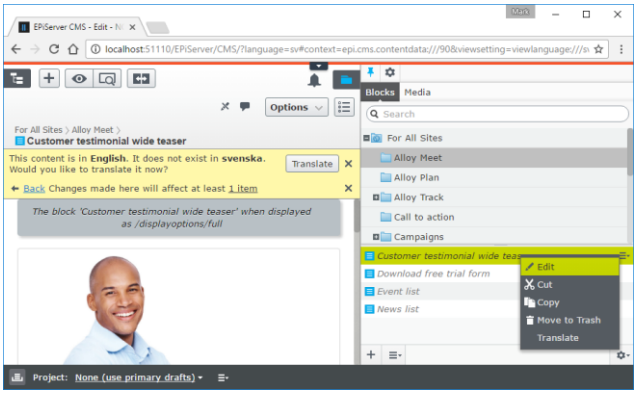
```
public class StandardPage : SitePageData
{
    [Display(
        GroupName = SystemTabNames.Content,
        Order = 310)]
    [CultureSpecific]
    public virtual XhtmlString MainBody { get; set; }

    [Display(
        GroupName = SystemTabNames.Content,
        Order = 320)]
    public virtual ContentArea MainContentArea { get; set; }
}
```

2. Open **~/Models/Blocks/TeaserBlock.cs**. Note the **Heading**, **Text**, and **Image** properties are `[CultureSpecific]`, as shown in the following code snippet:

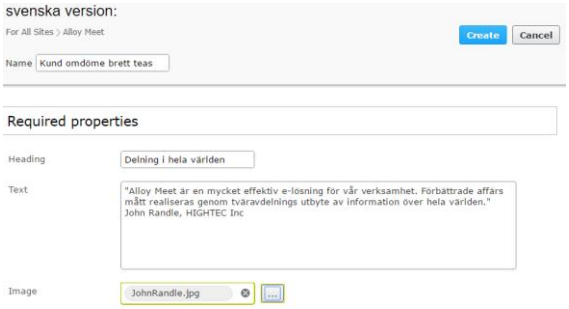
```
[CultureSpecific]
[Required(AllowEmptyStrings = false)]
[Display(
    GroupName = SystemTabNames.Content,
    Order = 1)]
public virtual string Heading { get; set; }
```

3. Edit the site and switch to the Swedish version of the Alloy Meet page.

4. In the **Assets** pane, view **Blocks**, open the **Alloy Meet** folder, and edit the **Customer testimonial wide teaser** block, as shown in the following screenshot:



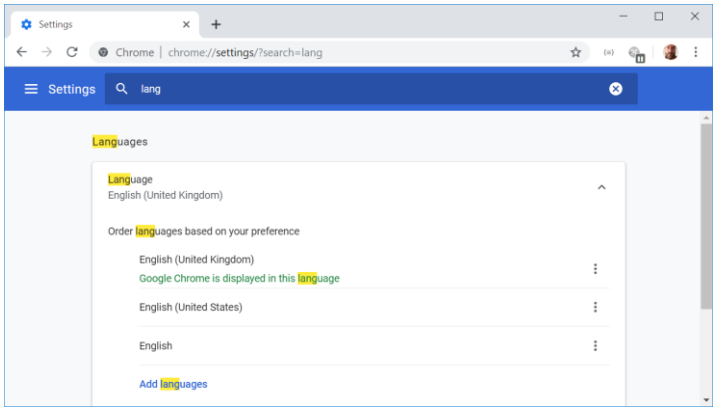
5. Translate the block into Swedish, and reuse the same image of John Randle, as shown in the following screenshot:



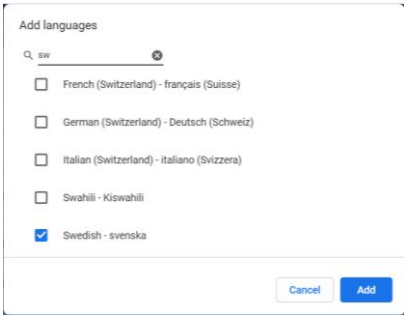
6. Publish the Swedish version of the block.
7. View the site as a visitor, and note the block is translated for Swedish page.

Configuring the site to detect language preference from the browser

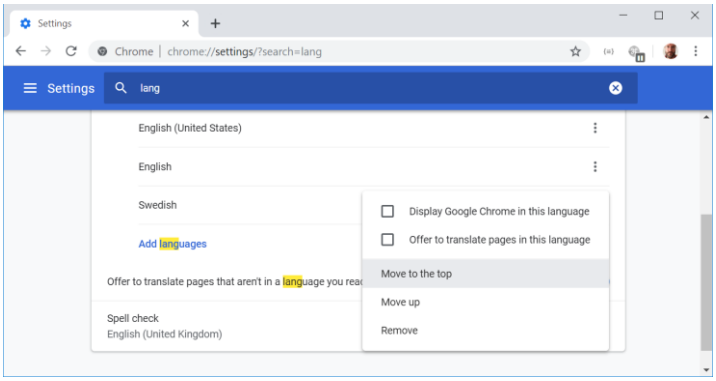
1. In **Google Chrome**, choose **Settings**, search for **lang**, and then expand the **Language** section, as shown in the following screenshot:



2. Click **Add languages**, search for and select **Swedish - svenska**, and then click **Add**, as shown in the following screenshot:

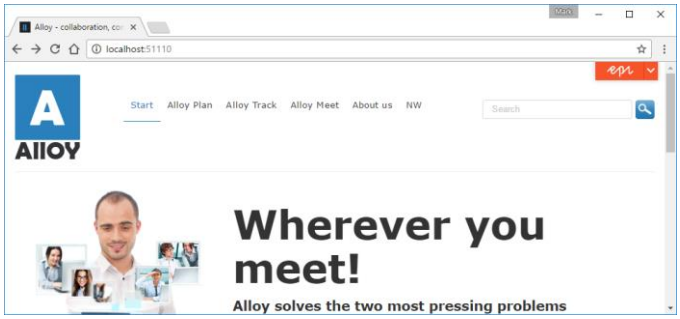


3. Click the three-dots menu for Swedish, and choose **Move to the top**, as shown in the following screenshot:



4. Close the **Settings** tab.

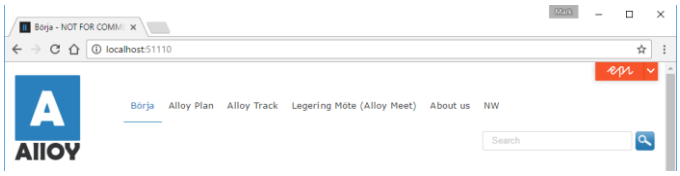
5. View the Start page as a visitor without a language code in the address bar, and the visitor sees the English version of the page, even though the browser is asking for Swedish, as shown in the following screenshot:



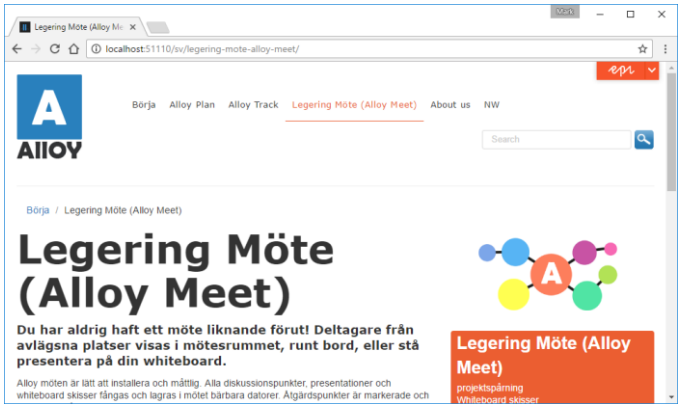
6. Close the browser and shut down the web server.
7. In `appsettings.Development.json`, add an entry in EPiServer | Cms for GlobalizationSettings to enable browser language preferences, as shown in the following configuration:

```
"EPiServer": {
  "Cms": {
    "GlobalizationSettings": {
      "UseBrowserLanguagePreferences": true
    }
```

8. Start the **AlloyDemo** website project.
9. View the site as a visitor again, this time you see Swedish by default, as shown in the following screenshot:



10. All links from the Start page will include `/sv/` to request Swedish pages, as shown in the following screenshot:



Automatically translating content using Optimizely Languages

Optimizely Languages add-on provides easy access to a single interface for managing multiple languages and translations of content, with a built-in feature for automated translation. No additional license fee is required for the add-on, and Microsoft Azure Translator can translate 2 million characters per month for free. It is an example of an add-on that registers itself as a gadget that can then be added to either the **Navigation** or the **Assets** panes.

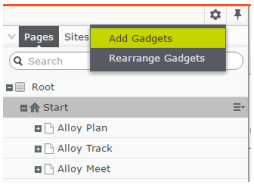
- **Optimizely Languages** is not tested on the Content Cloud developer certification exam. You can skip this topic if you do not have a Microsoft Azure account. Jump ahead to the task named **Localizing content types**, on page 66.

1. Open the solution with the **AlloyDemo** project.
2. Navigate to **Tools** | **NuGet Package Manager** | **Package Manager Console**.
3. Enter the following command:

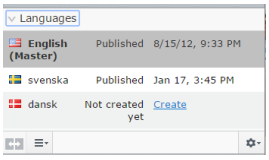
```
Install-Package EPiServer.Labs.LanguageManager -ProjectName AlloyDemo -Version 4.0.2
```

- You must specify the version number because the latest is 5.0.0 that only works with .NET 5 or later.

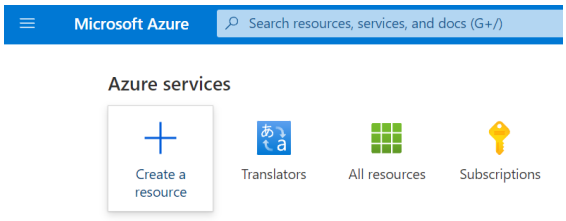
4. Start the site, and log in as **Admin**.
5. Navigate to **CMS** | **Edit**.
6. In the **Navigation** pane, on the **Settings** menu, click **Add Gadgets**, as shown in the following screenshot:



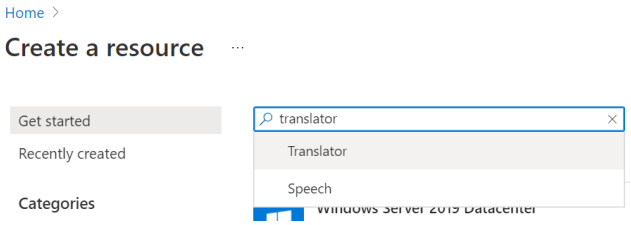
7. In the **Gadgets** picker, click **Languages**.
8. The **Languages** gadget will be added to the bottom of the **Navigation** pane, as shown in the following screenshot:



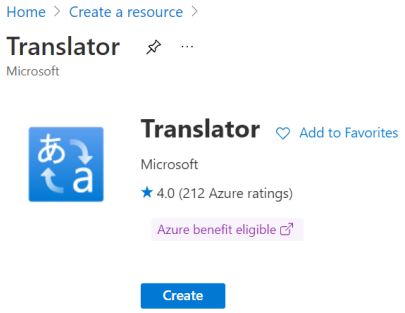
9. Sign into the Azure portal with a Microsoft Azure account: https://portal.azure.com/
10. Click **+ Create a resource**, as shown in the following screenshot:



11. Search for **Translator**, as shown in the following screenshot:



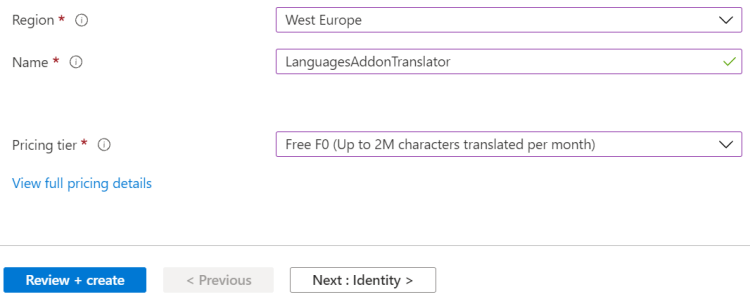
12. Click the **Create** button, as shown in the following screenshot:



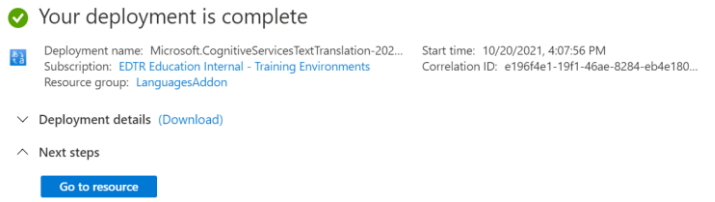
13. In the **Create Translator** page, set the following options:
    - **Subscription**: select your subscription
    - **Resource group**: create a new resource group named **CMSDevFun**.
    - **Resource group region**: select a suitable region e.g. **West Europe**.
    - **Resource region**: select a suitable region e.g. **West Europe**.
    - **Name**: **LanguagesAddonTranslator**
    - **Pricing tier**: F0 (2M Characters translated per month)

- F0 is the free tier.

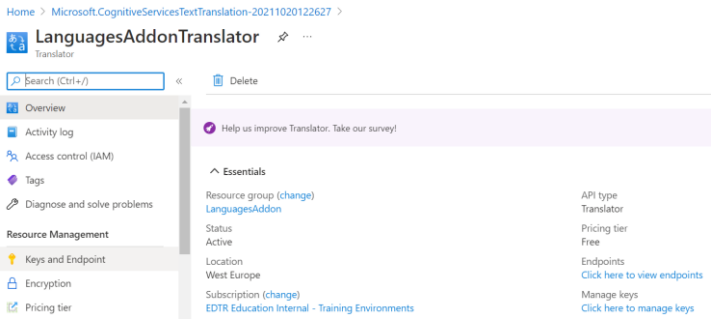
14. Click the **Review + Create** button, as shown in the following screenshot:



15. Review the summary of what will be created and then click **Create**.
16. Click the **Go to resource** button, as shown in the following screenshot:

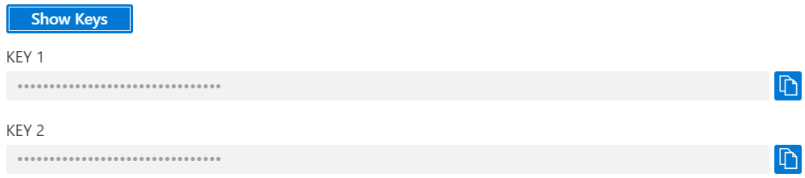


17. Click **Keys and Endpoint**, as shown in the following screenshot:

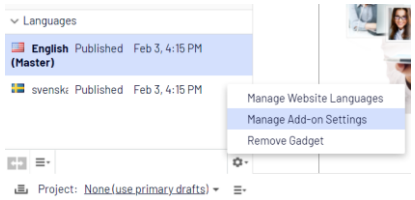


18. In the **Keys and Endpoint** blade, make a note of the **Location**. In my case, it is **westeurope**, as shown in the following screenshot:

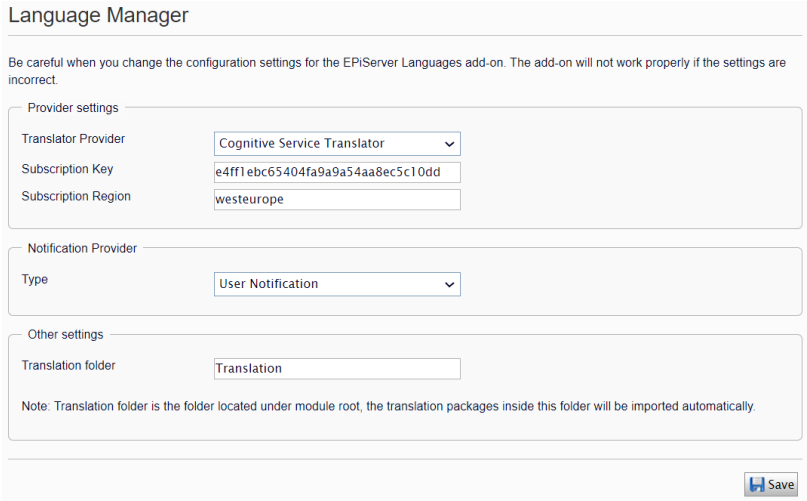
19. Click the **Show Keys** button, and then click the copy button next to one of the two keys, as shown in the following screenshot:



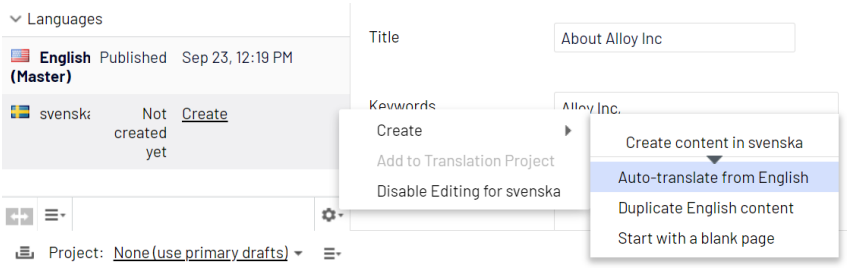
20. Back in Optimizely CMS, in the **Languages** gadget, click the **Settings** context menu, and then click **Manage Add-on Settings**, as shown in the following screenshot:



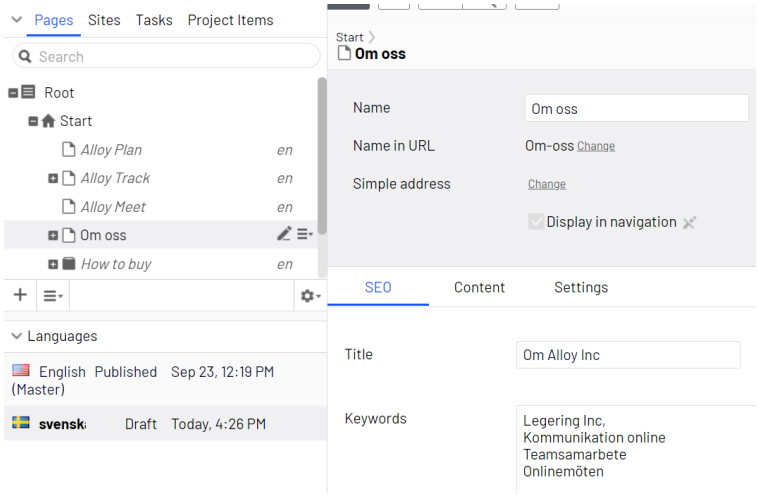
21. In **Admin** view, in **Language Manager**, set **Translator Provider** to **Cognitive Service Translator**, paste the key for the **Subscription Key**, and set the **Subscription Region** to match the location that you noted for the Translator resource, as shown in the following screenshot:



22. Click **Save**.
23. Navigate to **Edit** view and select **About us** in the **Pages** tree.
24. In **Languages**, for **svenska**, click the context menu (do NOT click the **Create** link!), then click the **Create** menu, and then **Auto-translate from English**, as shown in the following screenshot:

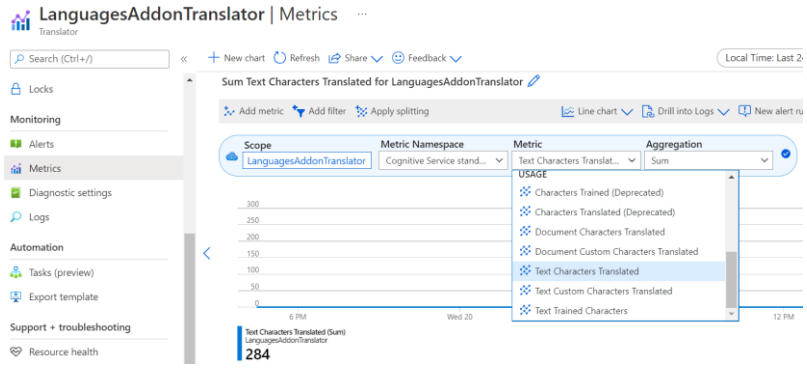


25. After a few seconds, any properties that are culture specific will be translated automatically, as shown in the following screenshot:



26. Publish the Swedish page.

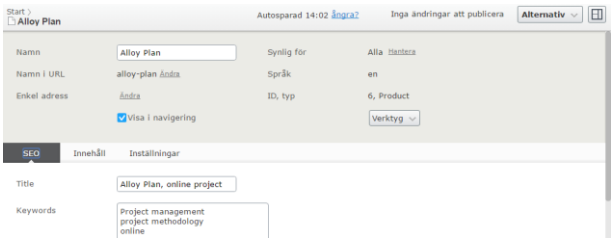
27. Back in Azure portal, note the **Monitoring | Metrics** section, where you can select a chart of **Text Characters Translated**, and see that 284 characters were translated, as shown in the following screenshot:



- You can review the Optimizely Languages user guide online: https://support.optimizely.com/hc/en-us/articles/4413199657741-Optimizely-Languages-add-on

Localizing content types

If the editor has chosen Swedish as their personal language and they edit the product page in **All Properties** view, Episerver localizes as much of the user experience labelling as possible, but it cannot automatically localize things like the page type name, custom group (tab) names like **SEO**, and custom property names like **Title** and **Keywords**, as shown in the screenshot:



Defining the localization text values

1. In **~/Resources/Translations**, copy and paste **ContentTypeNames.xml** by pressing *Ctrl* + *C*, then *Ctrl* + *V*.
2. Rename the copied file to **ContentTypesNames-sv.xml**.

- The name of these XML files does not matter. We are using `-sv` in the filename just for our use as identification. The important thing is the `id="sv"` inside the XML file as shown below.

3. Find the **<language>** element and modify it as follows:

```
<language name="Svenska" id="sv">
```

4. Find the **<productpage>** element and modify it as follows:

```
<productpage>
  <name>Produkt</name>
  <description>Används för att presentera en specifik produkt</description>
</productpage>
```

5. In **~/Resources/Translations**, copy and paste **PropertyNames.xml**.
6. Rename the copy to **PropertyNames-sv.xml**.
7. Find the **<language>** element and modify it as follows:

```
<language name="Svenska" id="sv">
```

8. Find the **<meta...>** elements and modify them as follows:

```
<metadescription>
  <caption>Sidbeskrivning</caption>
  <help>Används som meta beskrivning och allmänt som en ingress</help>
</metadescription>
<metakeywords>
  <caption>Nyckelord</caption>
</metakeywords>
<metatitle>
  <caption>Titel</caption>
</metatitle>
```

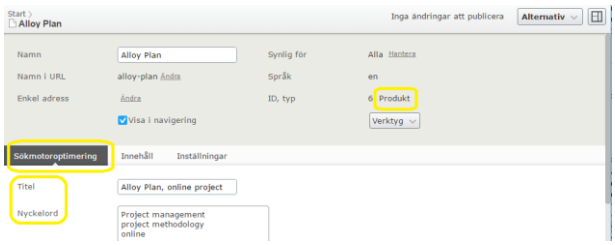
9. In **~/Resources/Translations**, copy and paste **GroupNames.xml**.
10. Rename the copy to **GroupNames-sv.xml**. Modify it as follows.

```
<?xml version="1.0" encoding="utf-8" ?>
<languages>
  <language name="Svenska" id="sv">
    <headings>
```

```
        <heading name="Contact">
          <description>Kontakta</description>
        </heading>
        <heading name="Default">
          <description>Standard</description>
        </heading>
        <heading name="News">
          <description>Nyheter</description>
        </heading>
        <heading name="Products">
          <description>Produkter</description>
        </heading>
        <heading name="MetaData">
          <description>Sökmotoroptimering</description>
        </heading>
        <heading name="SiteSettings">
          <description>Webbplatsinställningar</description>
        </heading>
        <heading name="Specialized">
          <description>Specialiserad</description>
        </heading>
      </headings>
    </language>
  </languages>
```

Viewing the localizations

1. Start the site, and log in as **Admin**.
2. Edit the **Alloy Plan** page, and switch to **All Properties** view.
3. Note the parts of the user interface that are now localized into Swedish that weren't before: the page type, the group name/tab, and the property names, as highlighted in the following screenshot:



4. Close the browser.

Localizing views for visitors

1. In the **AlloyDemo** project, expand the **Resources/Translations** folder, and open **Views.xml**.
2. Add an element inside the `<language name="English" id="en">` element, as shown in bold in the following markup:

```
<language name="English" id="en">
  <productpage>
    <changed>Changed:</changed>
  </productpage>
```

3. After the closing language element, add a new language element for the Swedish translation, as shown in the following markup:

```
<language name="Swedish" id="sv">
  <productpage>
```

```
    <changed>ändrats:</changed>
  </productpage>
</language>
```

4. In the **AlloyDemo** project, expand the **Views/ProductPage** folder, and open **Index.cshtml**.
5. Inside the `<h1>` element that outputs the name of the page, add a `<small>` element that outputs the **Changed** property formatted as a long date string, as shown in the following markup:

```
<h1 @Html.EditAttribtes(x => x.CurrentPage.PageName)>
    @Model.CurrentPage.PageName
    <small>@Html.Translate("/productpage/changed")
    @(Model.CurrentPage.Changed.ToLongDateString())</small>
</h1>
```

6. Start the site, navigate to **Alloy Meet** page, and note the **Changed** output is translated and formatted using **US English** culture, as shown in the following screenshot:

7. Change to the Swedish site, and note the **Changed** date is translated and formatted using English format, as shown in the following screenshot:

8. Close the browser.

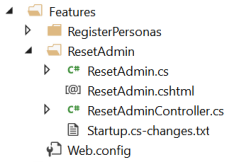
Exercise A6 – Resetting the Admin account

In this exercise, you will add functionality to reset the Admin account (if necessary).

Prerequisites: complete Exercise A1.

Adding the reset admin feature

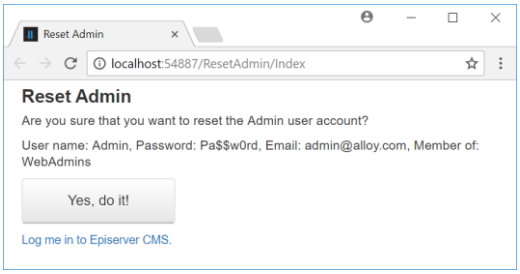
1. If you haven't done so already, extract the folders and files in **cmsdevfun-exercisefiles.zip**.
2. Drag and drop the **\cmsdevfun-exercisefiles\Module A\A6\Features** folder into the **AlloyDemo** project.
3. Expand the **Features** folder and review the files included, as shown in the following screenshot:



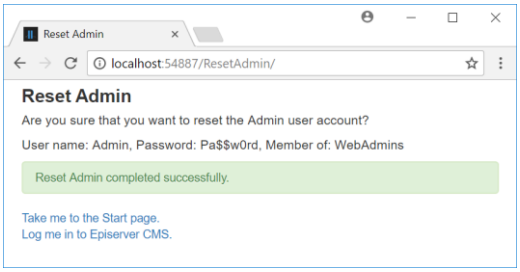
   a. **ResetAdmin.cs**: a static class to enable the feature.
   b. **ResetAdmin.cshtml**: a Razor file for the user interface of the feature.
   c. **ResetAdminController.cs**: a controller that performs the work of the feature.
4. Open **ResetAdminController.cs** file and modify the username and password if you want.
5. Open the **Startup.cs-changes.txt** file and copy and paste the statements into the appropriate place in the **Startup.cs** file.

Resetting the admin account

1. Start the **AlloyDemo** website.
2. On the **Reset Admin** page, click **Yes, do it!**, as shown in the following screenshot:



3. On the **Reset Admin** page, click **Log me in to Episerver CMS**, as shown in the following screenshot:



4. Log in using the new username and password.
5. Close the browser.

Disabling admin reset

1. Open the **Startup.cs** file.
2. Comment out or delete the statement that calls **UseResetAdmin()**.
3. Save changes and close the file.

Exercise A7 – Setting up Foundation – Frontline Services

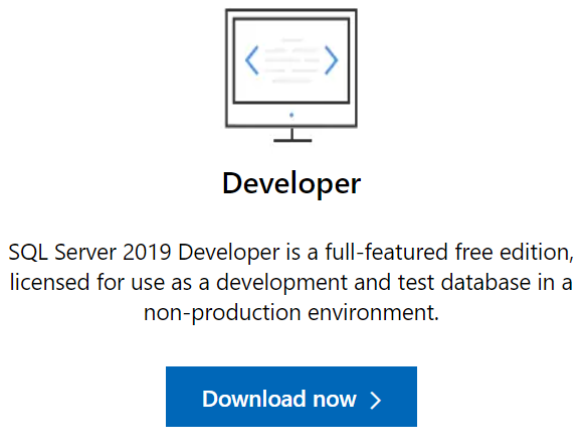
In this exercise, you will review the Foundation for CMS 12 website project.

Prerequisites: none.

Downloading and installing SQL Server and its Management Studio

In this task, you will download and install Microsoft SQL Server and its SQL Server Management Studio tool. This is because the **Foundation – Frontline Services** website project works best using the full version of SQL Server as its database server.

1. Start your favorite web browser.
2. Navigate to the download page for SQL Server: https://www.microsoft.com/en-us/sql-server/sql-server-downloads
3. Scroll down the page and in the **SQL Server 2019 Developer** section, click **Download now**, as shown in the following screenshot:



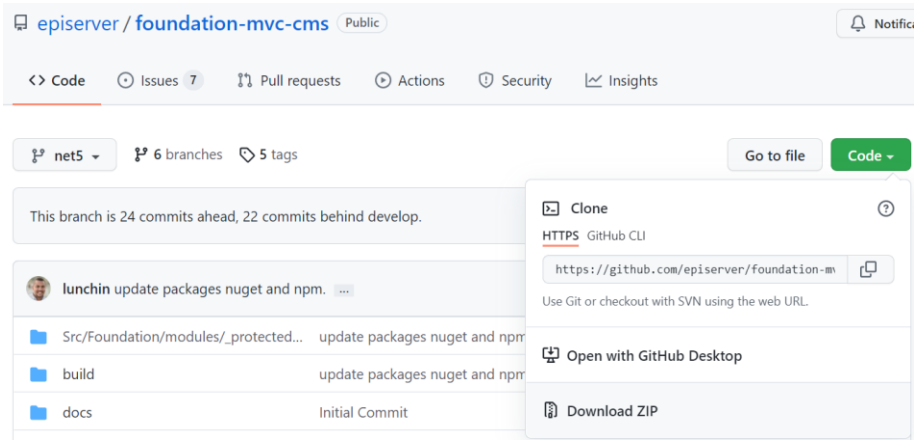
4. Run **SQL2019-SSEI-Dev.exe**.
5. Install SQL Server 2019 with default options including a local server name of . (dot)
6. Install SQL Server Management Studio.

Downloading a sample website project

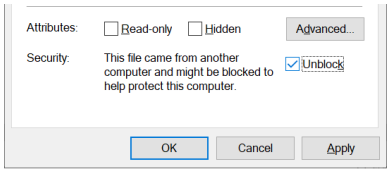
In this task, you will download a sample website project for a fictional organization named **Frontline Services** that is built using Optimizely CMS 12 on .NET 5.

1. Start your favorite web browser.
2. Navigate to the Optimizely GitHub repository for a sample starter CMS website project: https://github.com/episerver/foundation-mvc-cms/tree/net5

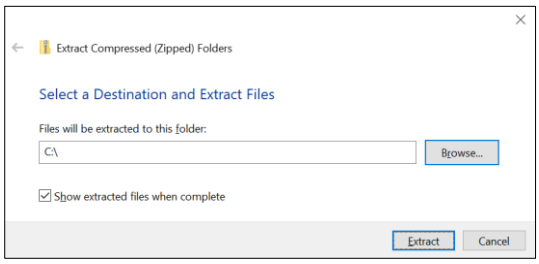
3. Click the green **Code** button and then click **Download ZIP**, as shown in the following screenshot:



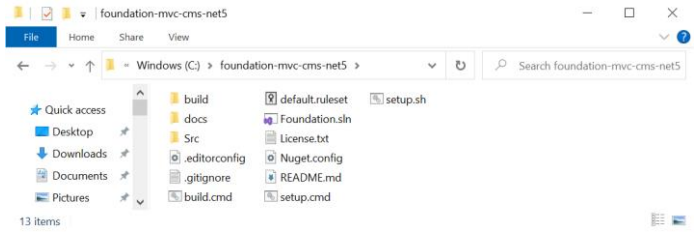
4. In your download folder, right-click the `foundation-mvc-cms-net5.zip` file, choose **Properties**, click **Unblock**, and then click **OK**, as shown in the following screenshot:



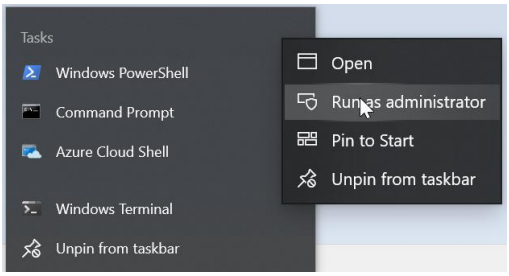
5. Right-click the `foundation-mvc-cms-net5.zip` file, choose **Extract All...**, set the folder to `C:\`, and then click **Extract**, as shown in the following screenshot:



6. Open the folder in **File Explorer**, as shown in the following screenshot:



7. Run **Command Prompt** or **Windows Terminal** as administrator, as shown in the following screenshot:



8. At the command-line, enter the command to change to the folder:

```
cd C:\foundation-mvc-cms-net5\
```

9. At the command-line, enter the command to execute the setup script:

```
./setup.cmd
```

10. Enter values for:

    a. App name: `Foundation`.

    b. SQL server name: `.` or press *Enter*.

    c. Press *Enter* to use your Windows account for authentication.

```
Enter your app name (required):Foundation
Enter your SQL server name (optional, press Enter for default (.) local server):
Enter your sqlcmd command (optional, press Enter for default (-E) windows auth):
```

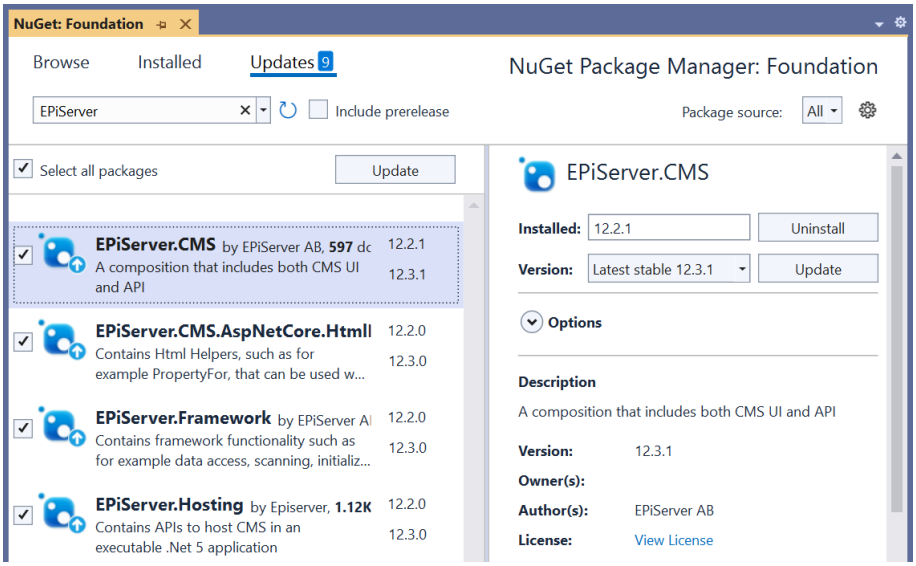
11. Press any key and then wait for the results, as shown in the following output:

```
#                          _____________                          #
#                         |_____________| ___                     #
#                         |             |/ _ \                    #
#                         |               | | |                   #
#                         |     EPI       |_| |                   #
#                       ___|             |\___/                   #
#                      /    \___________/    \                    #
#                      \_____________________/                    #
#                                                                 #
###################################################################
## Building Foundation please check the Build\Logs directory if you receive errors
## Gettting MSBuildPath ##
msbuild.exe path: C:\Program Files\Microsoft Visual
Studio\2022\Community\MSBuild\Current\Bin\MSBuild.exe
## NPM Install ##

> core-js@3.6.5 postinstall C:\foundation-mvc-cms-
net5\Src\Foundation\node_modules\core-js
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling
JavaScript standard library!

The project needs your help! Please consider supporting of core-js on Open
Collective or Patreon:
> https://opencollective.com/core-js
Also, the author of core-js ( https://github.com/zloirock ) is looking for a good
job -)
> @fortawesome/fontawesome-free@5.15.4 postinstall C:\foundation-mvc-cms-
net5\Src\Foundation\node_modules\@fortawesome\fontawesome-free
```

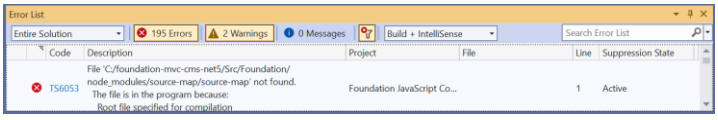
```
Font Awesome Free 5.15.4 by @fontawesome - https://fontawesome.com
added 388 packages in 22.712s

> Foundation-mvc-cms@1.0.0 dev C:\foundation-mvc-cms-net5\Src\Foundation
assets by chunk 2.79 MiB (auxiliary name: main)
  assets by path ../vendors/ 2.79 MiB 15 assets
  assets by info 1.06 KiB [immutable]
    assets by path *.woff2 222 bytes 3 assets
    assets by path *.woff 219 bytes 3 assets
    assets by path *.eot 216 bytes 3 assets
    assets by path *.svg 216 bytes 3 assets
    assets by path *.ttf 216 bytes 3 assets
assets by chunk 1.36 MiB (name: main)
  asset main.min.js 1020 KiB [emitted] (name: main) 1 related asset
  asset ../scss/main.min.css 376 KiB [emitted] (name: main) 1 related asset

The project needs your help! Please consider supporting of core-js on Open
Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a good
job -)


> @fortawesome/fontawesome-free@5.15.4 postinstall C:\foundation-mvc-cms-
net5\Src\Foundation\node_modules\@fortawesome\fontawesome-free
> node attribution.js

Font Awesome Free 5.15.4 by @fontawesome - https://fontawesome.com
License - https://fontawesome.com/license/free (Icons: CC BY 4.0, Fonts: SIL OFL
1.1, Code: MIT License)

added 388 packages in 22.712s

> Foundation-mvc-cms@1.0.0 dev C:\foundation-mvc-cms-net5\Src\Foundation
> webpack --config webpack.dev.js

assets by chunk 2.79 MiB (auxiliary name: main)
  assets by path ../vendors/ 2.79 MiB 15 assets
  assets by info 1.06 KiB [immutable]
    assets by path *.woff2 222 bytes 3 assets
    assets by path *.woff 219 bytes 3 assets
    assets by path *.eot 216 bytes 3 assets
    assets by path *.svg 216 bytes 3 assets
    assets by path *.ttf 216 bytes 3 assets
assets by chunk 1.36 MiB (name: main)
  asset main.min.js 1020 KiB [emitted] (name: main) 1 related asset
  asset ../scss/main.min.css 376 KiB [emitted] (name: main) 1 related asset
Entrypoint main 1.36 MiB (4.48 MiB) = ../scss/main.min.css 376 KiB main.min.js 1020
KiB 32 auxiliary assets
orphan modules 1020 KiB (javascript) 1.06 KiB (asset) 1.02 KiB (runtime) [orphan] 41
modules
runtime modules 937 bytes 4 modules
modules by path ./node_modules/ 854 KiB (javascript) 59.2 KiB (css/mini-extract) 97
modules
modules by path ./wwwroot/ 22.9 KiB (javascript) 317 KiB (css/mini-extract)
  modules by path ./wwwroot/js/ 22.8 KiB 8 modules
  modules by path ./wwwroot/scss/*.scss 50 bytes (javascript) 317 KiB (css/mini-
extract)
    ./wwwroot/scss/main.scss 50 bytes [built] [code generated]
    css ./node_modules/css-loader/dist/cjs.js!./node_modules/sass-
loader/dist/cjs.js??ruleSet[1].rules[0].use[2]!./wwwroot/scss/main.scss 317 KiB
[built] [code generated]
modules by path ./Features/ 25 KiB
  modules by path ./Features/Search/*.js 12.5 KiB 2 modules
```

```
  + 3 modules
webpack 5.67.0 compiled successfully in 9060 ms
## Clean and build ##
## sqlcmd -S . -E ##
## Dropping databases ##
## Dropping user ##
Tool 'episerver.net.cli' is already installed.
---------------------------------------
- Optimizely database cli tool, v2.0.0 -
---------------------------------------

[17:29:08 INF] Running create-cms-database command.
[17:29:09 WRN] The username Foundation.CmsUser already exists on the server, no need
to create.
[17:29:10 INF] create-cms-database has completed.
Run resetup.cmd to resetup solution
Press any key to continue . . .
```

12. Close **Command Prompt** or **Windows Terminal**.
13. Open **Foundation.sln** using Visual Studio.
14. In **Solution Explorer**, right-click **Dependencies** and select **Manage NuGet Packages...**
15. Select the **Updates** tab, for **Package source**, select **All**, enter **EPiServer** in the search box, check the **Select all packages** box, and then click the **Update** button, as shown in the following screenshot:



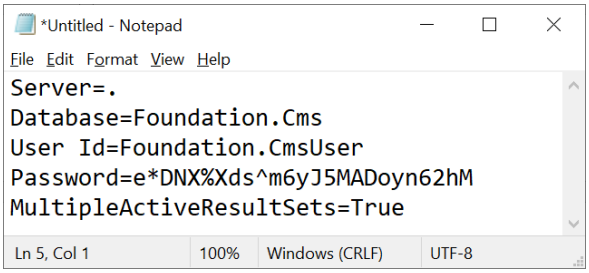
16. In the **Preview Changes** dialog box, review which packages are about to be updated, and then click **OK**.
17. In the **License Acceptance** dialog box, click **I Accept**.
18. Navigate to **Build** | **Rebuild Solution**.
19. Ignore the errors and warnings, as shown in the following screenshot:



20. Open **appsettings.json** and note the database connection string, as shown in the following markup:

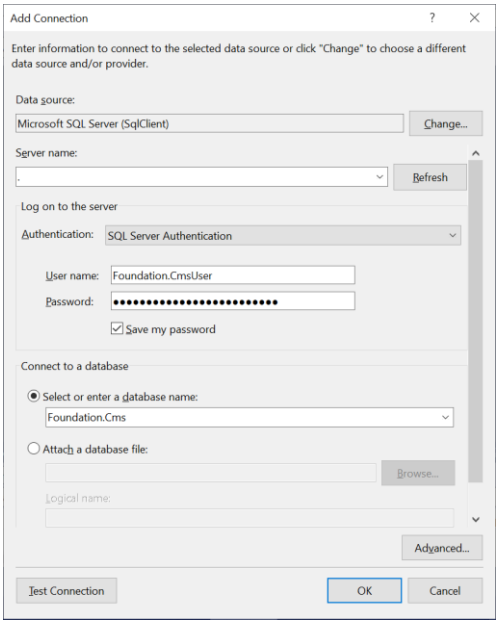
```
{
  "ConnectionStrings": {
    "EPiServerDB": "Server=.;Database=Foundation.Cms;User
Id=Foundation.CmsUser;Password=e*DNX%Xds^m6yJ5MADoyn62hM;MultipleActiveResultSets=Tr
ue"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "EPiServer": {
    "Find": {
      "DefaultIndex": "changeme",
      "ServiceUrl": "https://changeme.com/",
      "TrackingSanitizerEnabled": true,
      "TrackingTimeout": 30000,
      "UI": {}
    }
  }
}
```

21. Run **Notepad**, copy and paste the database connection string, and format it so that each parameter is on its own line (delete the semi-colon separators), as shown in the following screenshot:

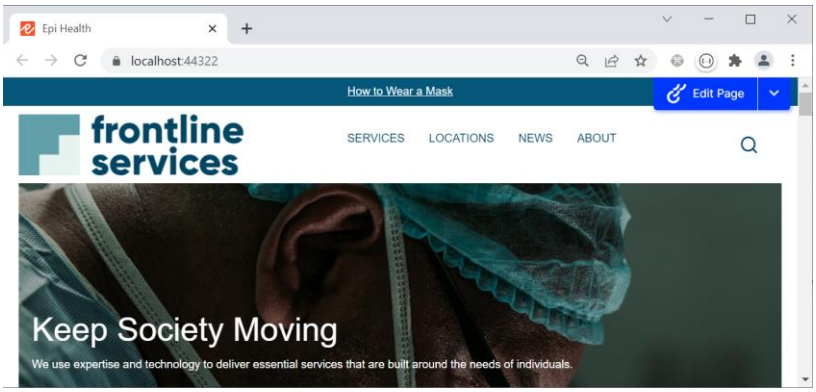


22. In Visual Studio, navigate to **View** | **Server Explorer**.
23. Right-click **Data Connections** and select **Add Connection...**

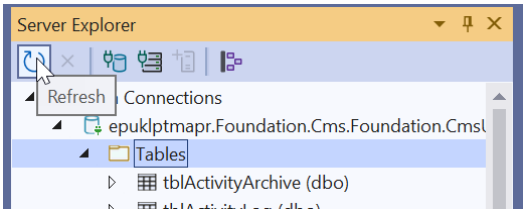
24. In the Add Connection dialog box, for **Authentication** select **SQL Server Authentication** and **Save my password**, and then copy and paste the details for **Server name**, **User name**, **Password**, and **Database name** from Notepad, as shown in the following screenshot:



25. Click **Test Connection**, and then click **OK**.
26. In **Server Explorer**, click to expand **Tables** and note that there are no tables! The database schema is not created until you start the website project for the first time.
27. Navigate to **Debug** | **Start Without Debugging** or press *Ctrl* + *F5*.
28. Wait for the website to start, as shown in the following screenshot:



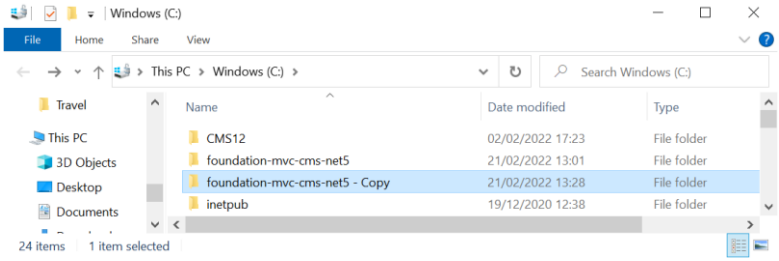
29. Navigate to SERVICES, NEWS, ABOUT (but not LOCATIONS) to become familiar with the website.
30. Close the browser.
31. In **Server Explorer**, click **Refresh**, and note that CMS tables have now been created, as shown in the following screenshot:



32. Close the database connection and then close **Server Explorer**.

Optional: create a backup copy of the Foundation – Frontline Services solution folder

Close the solution in Visual Studio and then in File Explorer, copy and paste the folder.



Remember that its database is stored in SQL Server so is not included in this backup.

- Congratulations! You have set up **Foundation – Frontline Services**. You can use this for demonstrating commonly used add-ons as well as for a starter solution for your own website projects.

Exercise A8 – Create a custom editing experience for date-only pickers using Dojo

In this exercise, you will swap a built-in editing experience for a more appropriate one.

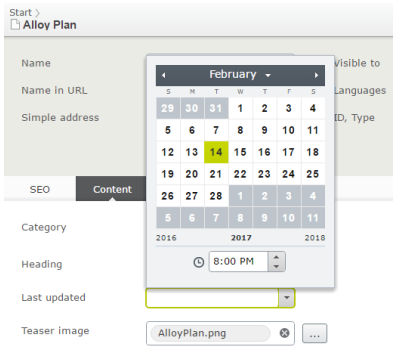
Prerequisites: complete Exercise A1.

Reviewing the date and time picker

1. Open the solution with the **AlloyDemo** project.
2. In **~\Models\Pages**, open **ProductPage.cs**.
3. Add a property to store when the product was last updated, as shown in the following code:

```
[Display(Name = "Last updated", Order = 20)]
public virtual DateTime? LastUpdated { get; set; }
```

4. Start the site, and log in as **Admin**.
5. Edit the **Alloy Plan** page, switch to **All Properties** view, and click the **Content** tab.
6. When you change the **Last updated** property, a Dojo date and time picker is used, as shown in the screenshot:



- Most of the default **dijits** (Dojo "form widgets") can be swapped in as replacements for compatible Episerver properties. All you need to do is to set the proper **ClientEditingClass**. Read about all the available "form widgets": http://dojotoolkit.org/reference-guide/1.10/dijit/form.html#dijit-form

Creating a date-only picker

1. Open **~/Globals.cs**.
2. Add a `string` constant named `DateOnly` to the `SiteUIHints` class, as shown in the following code:

```
public static class SiteUIHints
{
    public const string City = "City";
    public const string Contact = "contact";
    public const string DateOnly = "dateonly";
    public const string Strings = "StringList";
}
```

3. In **~\Business\EditorDescriptors**, add a class named **DateOnlyEditorDescriptor.cs**.
4. Modify the statements as shown in the following code:

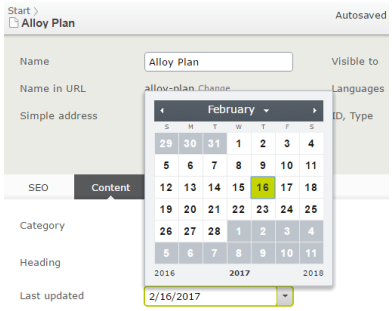
```
using EPiServer.Shell.ObjectEditing;
using EPiServer.Shell.ObjectEditing.EditorDescriptors;

namespace AlloyDemo.Business.EditorDescriptors
{
    [EditorDescriptorRegistration(TargetType = typeof(DateTime),
        UIHint = Globals.SiteUIHints.DateOnly)]
    [EditorDescriptorRegistration(TargetType = typeof(DateTime?),
        UIHint = Globals.SiteUIHints.DateOnly)]
    public class DateOnlyEditorDescriptor : EditorDescriptor
    {
```

```
        public override void ModifyMetadata(ExtendedMetadata metadata,
            IEnumerable<Attribute> attributes)
        {
            ClientEditingClass = "dijit/form/DateTextBox";
            base.ModifyMetadata(metadata, attributes);
        }
    }
}
```

5. Open **~\Models\Pages\ProductPage.cs**.
6. Apply `[UIHint]` to the **LastUpdated** property to activate the editor descriptor that you just created, as shown in the following code:

```
[Display(
    Name = "Last updated",
    GroupName = SystemTabNames.Content,
    Order = 20)]
[UIHint(Globals.SiteUIHints.DateOnly)]
public virtual System.DateTime? LastUpdated { get; set; }
```

7. Start the site, and log in as **Admin**.
8. Edit the **Alloy Plan** page.
9. Modify the **Last updated** property, and note that the time part is now hidden, as shown in the screenshot:



Read more about Dojo

http://marisks.net/2014/04/11/episerver-writing-dojo-widget/

https://robertlinde.se/episerver%207.5/plugin/custom%20property/dojo/2014/05/10/custom-episerver-properties-with-dojo.html

https://andersnordby.wordpress.com/2014/10/24/creating-a-custom-property-with-a-dojo-widget/

https://www.david-tec.com/2014/08/EPiServer-Dojo-Useful-links/

http://azanganeh.com/blog/2016/10/30/episerver-custom-property-in-simple-steps-by-step-example/

Exercise A9 – Identifying website features

In this exercise, you will identify the features that public websites built using Optimizely Content Cloud (CMS) have implemented.

Prerequisites: none.

Choose one of the following public websites created with Optimizely Content Cloud (CMS), and identify the following features:

- Identify what markup is part of a shared layout.
- Identify at least three page types and list their likely properties including data types.
- Identify at least three block types and list their likely properties including data types.
- Identify at least one interactive feature, like a form that the visitor can interact with.

Optimizely Content Cloud (CMS) websites

- https://www.roehampton.ac.uk/
- https://www.mazdausa.com/
- https://www.absolut.com/
- https://www.wexphotovideo.com/
- https://www.southwesternrailway.com/
- https://www.gatwickairport.com/

Module B – Defining Content Types

Goal

The overall goal of the exercises in this module is to implement typical examples of content types with templates and layouts while following good practice. You will:

1. Set up an **Empty CMS** website and define a **Start** page type with a page template.
2. Define media types to handle generic files, image files, and SVG files.
3. Create shared layouts for page templates, follow good practice to define a base page type and a base page controller that uses constructor parameter injection for dependencies, and define a page view model for use in layouts, views, and controllers.
4. Define a **Standard** page with a **MainBody** property and an alternative layout, a **Product** page with **UniqueSellingPoints** property and a nested layout, and then add a navigation menu to the root layout.

Exercise B1 – Setting up the AlloyTraining website

In this exercise, you will set up an Empty CMS website and update it to use the latest version of Optimizely CMS. You will:

- Add Bootstrap files.
- Add helper classes used in later exercises to save you time.
- Define a Start page type.
- Create a page template for the Start page type.
- Localize the Start page type for editors who speak Swedish but not English

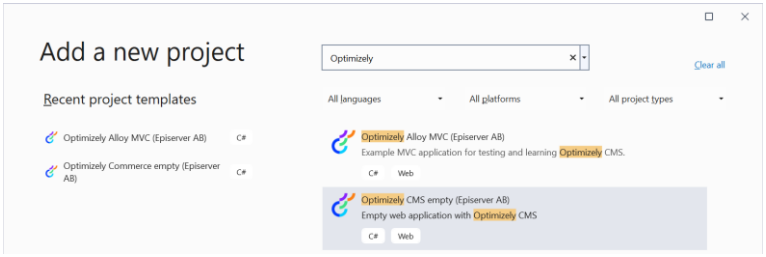
Prerequisites: Microsoft Visual Studio 2022 or later, or Visual Studio Code, and Optimizely project templates.

Creating AlloyTraining from the epi-cms-empty project template

1. In Visual Studio 2022, open your previous solution, and navigate to **File** | **Add** | **New Project...**

- You can use a new solution if you prefer.

2. In the search box, enter **Optimizely**, select the **Optimizely CMS empty (Episerver AB)** project template, and then click **Next**.

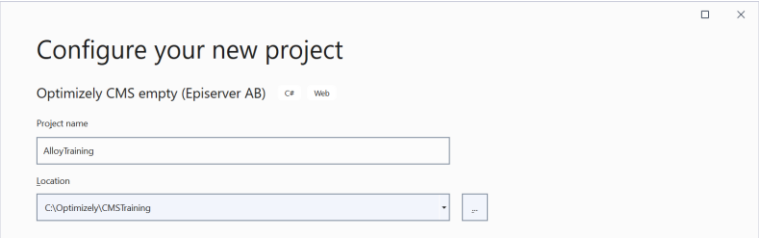


3. Enter the following options:

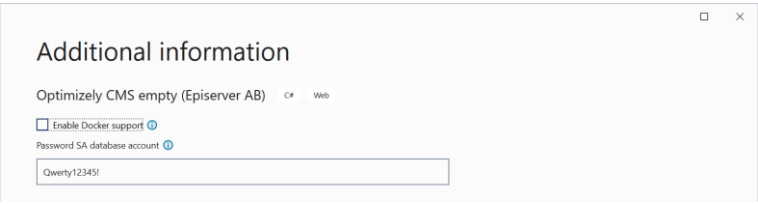
- You MUST name your project **AlloyTraining**. The solution classes have been written assuming that project name and so all the namespaces will use it, e.g., **AlloyTraining.Models.Pages.StartPage**, and so on.

    - Project name: **AlloyTraining**

- Location: **C:\Optimizely\CMSTraining** (or whatever location you used before).



4. Click **Next**.
5. Leave the options for Docker and the SA database password as their defaults, as shown in the screenshot, and then click **Create**.



6. Wait for NuGet packages to restore, as shown in the following screenshot:

7. In **Solution Explorer**, right-click the **CMSTraining** solution and click **Set StartUp Projects**.
8. Click **Current selection**, and then click **OK**.

- You will now be able to quickly switch between running multiple projects by selecting a project in Solution Explorer and pressing *Ctrl + F5*.

Installing the Forms add-on

Commented [BG1]: Using the EPiServer.CMS 12.18.0 version on 9/9/23, this fails due to package version constraints, probably need to update the project first.

1. In Visual Studio, navigate to **Tools** | **NuGet Package Manager** | **Package Manager Console**.
2. In **Package Manager Console**, set these two options:
   a. Package source: **All**
   b. Default project: **AlloyTraining**
3. Enter the following command to install Optimizely Forms:

```
Install-Package EPiServer.Forms -ProjectName AlloyTraining
```

Disabling null checks

.NET 6 enables null check warnings by default. Optimizely CMS APIs have not yet been annotated to indicate how to perform static code analysis for nullability. To avoid seeing those warnings, we can disable nullable for the project:

1. Open the **AlloyTraining.csproj** project file.
2. Change the `Nullable` setting to `disable`, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
```

```
    <Nullable>disable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="EPiServer.CMS" Version="12.6.0" />
    <PackageReference Include="EPiServer.Forms" Version="5.1.0" />
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resources\Translations\**\*" />
  </ItemGroup>
</Project>
```

3. Save changes and close the project file.

Controlling the port number

By default, both the Alloy MVC and Empty CMS project templates set the port number to 5000 in the `applicationUrl`. This means you cannot run both projects at the same time unless we change one of the projects to use a different port number:

1. In **AlloyTraining**, in the **Properties** folder, open **launchSettings.json**.
2. In the `applicationUrl` setting, modify the port number to 5001 (**AlloyDemo** should continue to use 5000), as shown in the following settings:

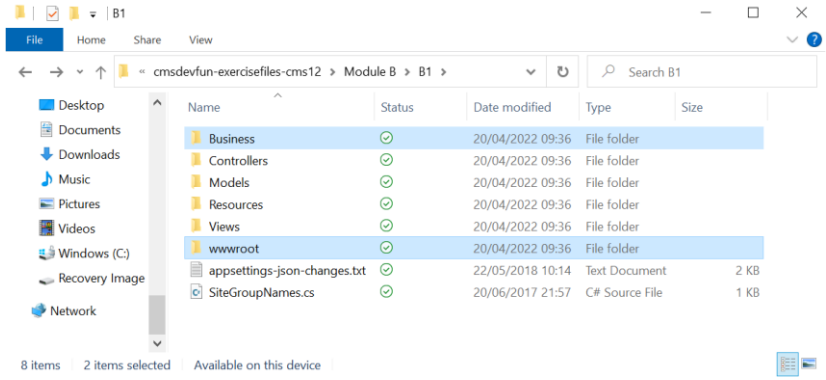
```
{
  "profiles": {
    "AlloyTraining": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001/",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

3. Optionally, change the `launchBrowser` setting to `false` to prevent the errors when Visual Studio launches the browser faster than the Kestrel web server can get the website listening. But this will mean you must manually start the browser and navigate to https://localhost:5001/

Adding Bootstrap and Business logic

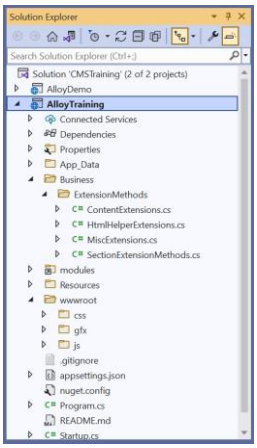
1. If you haven't already, then extract the folders and files in **cmsdevfun-exercisefiles.zip**.

2. Drag and drop, or copy, the **Business** and **wwwroot** folders from **cmsdevfun-exercisefiles\Module B\B1** to the root of the **AlloyTraining** project, as shown in the following screenshot:



- Make sure you drag and drop to the **AlloyTraining** project, NOT AlloyDemo!

3. In **Solution Explorer**, expand the **Business** and **wwwroot** folders, as shown in the following screenshot:



- If Visual Studio doesn't allow you to drag and drop, you can open the folder with File Explorer and copy and paste.

4. Note the following folders and files were added:
   - **Extension methods** for use later in the exercises to generate menus and so on.
   - **wwwroot** application files for stylesheets, JavaScript libraries, and graphics.
5. In **AlloyTraining.csproj**, make sure that there are no entries that would affect the **wwwroot** or **Business** folders and their subfolders and files, for example, remove entries like those shown in the following markup:

```
<ItemGroup>
  <None Include="wwwroot\css\bootstrap-collapse.js" />
  <None Include="wwwroot\js\bootstrap.js" />
  <None Include="wwwroot\js\jquery.js" />
```

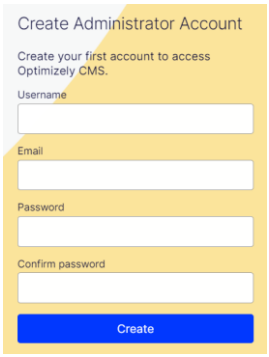
```
</ItemGroup>
```

6. Click **Build** | **Build Solution** to ensure the website compiles.

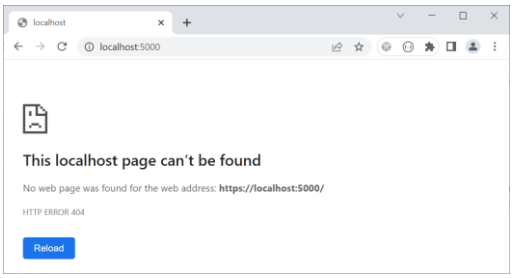
Testing the Empty CMS website

Now we can try running the website project:

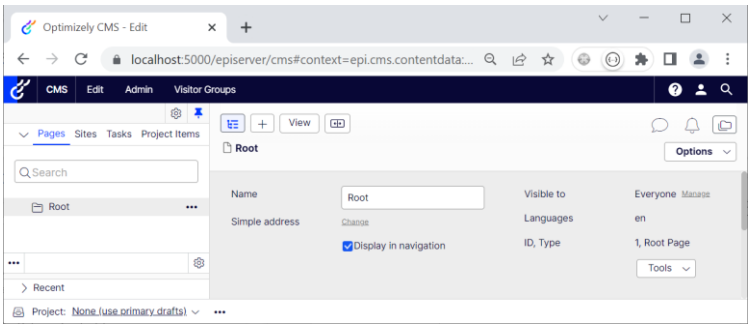
1. In **Solution Explorer**, click **AlloyTraining**.
2. Start the website by navigating to **Debug** | **Start Without Debugging** or press *Ctrl + F5*.
3. Note that you are redirected to the register page, as shown in the screenshot:



4. Enter suitable values:
    a. Username: `Admin`
    b. Email: `admin@alloy.com`
    c. Password: `Pa$$w0rd`
5. Click **Create**.
6. Note that you are redirected to the start page, which does not yet exist, so you will see a 404 message, as shown in the following screenshot:



7. In the browser address box, enter `/episerver/cms` and note you are switched to **Edit** view and the only content in the **Pages** tree is the **Root** page, as shown in the following screenshot:



8. Close the browser and shut down the web server (press *Ctrl + C* at the command line).

Defining a class of string constants for grouping content types

Content types can be grouped when shown as a list for the editors. You will start by defining a static class with `string` constants for the group names you will use in your site.

1. In the **AlloyTraining** project, add a new class file named **SiteGroupNames.cs**.
2. Modify the file, as shown in the following code:

```
namespace AlloyTraining
{
    public static class SiteGroupNames
    {
        public const string Specialized = "Specialized";
        public const string Common = "Common";
        public const string News = "News";
    }
}
```

- Remember that you can copy and paste or drag and drop solution files into your project to save time.

Defining a Start page type

1. In **AlloyTraining**, create a new folder named **Models**.
2. In **Models**, create a new folder named **Pages**.
3. In **Pages**, right-click and click **Open in Terminal**.
4. At the command line, get help about creating a content type using the `dotnet new` item template, as shown in the following command:

```
dotnet new epi-cms-contenttype --help
```

5. At the command line, create a page type with an appropriate class name and namespace using the appropriate `dotnet new` item template, as shown in the following command:

```
dotnet new epi-cms-contenttype -p:na AlloyTraining.Models.Pages -n
StartPage
```

6. Review the **StartPage** class, as shown in the following code:

```
using System;
using System.ComponentModel.DataAnnotations;
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;

namespace AlloyTraining.Models.Pages
{
  [ContentType(
    DisplayName = "StartPage",
    GUID = "BEA2D452-4DBA-4BA7-9AFC-FC7A1E4F8424",
    Description = "")]
  public class StartPage : PageData
  {
    [CultureSpecific]
    [Display(
      Name = "MyProperty",
      Description = "My property description",
      GroupName = SystemTabNames.Content,
      Order = 10)]
    public virtual string MyProperty { get; set; }
  }
}
```

7. In the [ContentType] attribute:
    - Change the **DisplayName** to **Start**.
    - Set the **GroupName** to **SiteGroupNames.Specialized** and the sort index (i.e. **Order**) within that group to **10**.
    - Set the **Description** of "The home page for a website with an area for blocks and partial pages."
8. In the body of the class, change name of the **MyProperty** property to **MainBody**, the data type to **XhtmlString**, and the **Description** to "The main body uses the XHTML-editor so you can insert, for example text, images, and tables."
9. Add a string **Heading** property with appropriate attribute values.

- **Order** values are used to sort properties within a tab group. It is good practice to use multiples of ten so that future properties can be added between existing ones.

10. Add a content area property with appropriate attribute values.

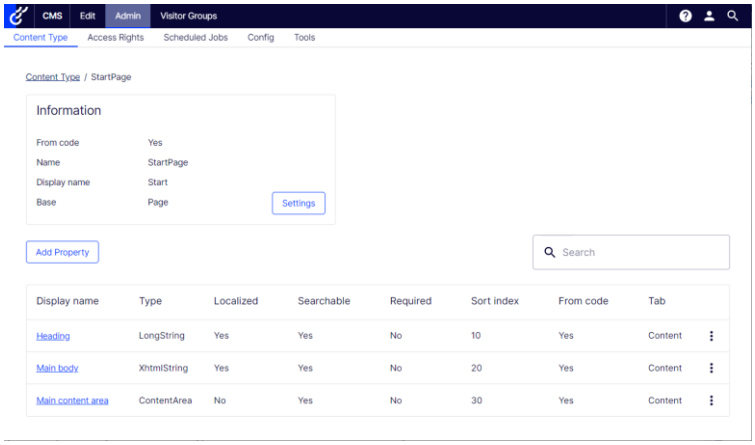
Your complete page type class should look something like the following code:

```
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;
using System.ComponentModel.DataAnnotations;

namespace AlloyTraining.Models.Pages
{
    [ContentType(DisplayName = "Start",
        GroupName = SiteGroupNames.Specialized, Order = 10,
        // your code will have a random GUID here
        Description = "The home page for a website with an area for blocks and partial
pages.")]
    public class StartPage : PageData
    {
        [CultureSpecific]
        [Display(Name = "Heading", Description =
            "If the Heading is not set, the page falls back to showing the Name.",
            GroupName = SystemTabNames.Content, Order = 10)]
        public virtual string Heading { get; set; }

        [CultureSpecific]
        [Display(Name = "Main body",
            Description = "The main body uses the XHTML-editor so you can insert, for
example text, images, and tables.",
            GroupName = SystemTabNames.Content, Order = 20)]
        public virtual XhtmlString MainBody { get; set; }

        [Display(Name = "Main content area",
            Description = "The main content area contains an ordered collection to
content references, for example blocks, media assets, and pages.",
            GroupName = SystemTabNames.Content, Order = 30)]
        public virtual ContentArea MainContentArea { get; set; }
    }
}
```

- **SystemTabNames.Content** is the default so setting the **GroupName** attribute to this value is optional. Code in this exercise book, and in **cmsdevfun-exercisefiles.zip**, will not have GUIDs set for content types so that you can use the solution code and it will match based on namespace and class name instead of GUID values.

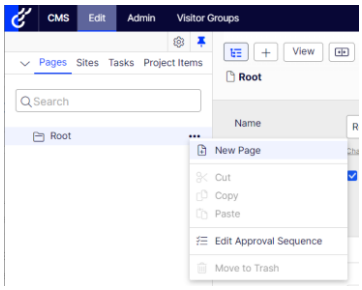
Creating a Start page instance

1. Start the website by navigating to **Debug** | **Start Without Debugging** or press *Ctrl + F5*.
2. Enter `/episerver/cms/` at the end of the address box.
3. Log in as **Admin**.
4. Navigate to **CMS** | **Admin** | **Content Type**, filter the table to only show **Page** types, click **Start**, and note that the synchronization process has registered your page type, including its properties, as shown in the following screenshot:

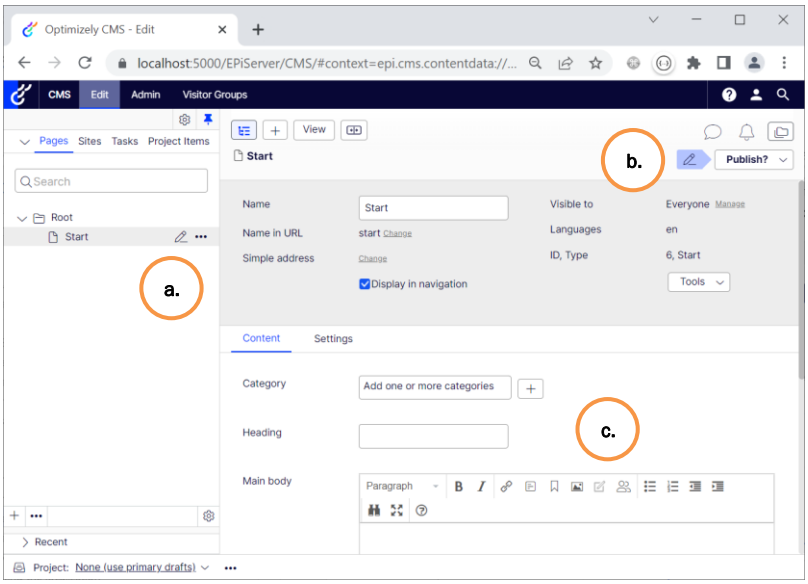


- Members of the virtual role **CmsAdmins** can modify many of the values that are initially set by your code attributes by clicking **Settings**. These changes will override your code, even if you subsequently change your code.

5. Navigate to **CMS** | **Edit**, expand the **Navigation** pane, and note it only contains a **Root** page.
6. Click the context menu for the **Root** page and then click **New Page**, as shown in the screenshot:
7. Enter the name **Start**, and then click **OK**.



8. Note the following:
    a. Pencil icon indicates a saved draft, so the page is not published yet and it won't be visible to visitors.
    b. Blue information icon warns that you must publish this page for visitors to see changes.
    c. The three custom properties, **Heading**, **MainBody**, and **MainContentArea**, are grouped and sorted under the **Content** tab in **All Properties** view, as shown in the following screenshot:



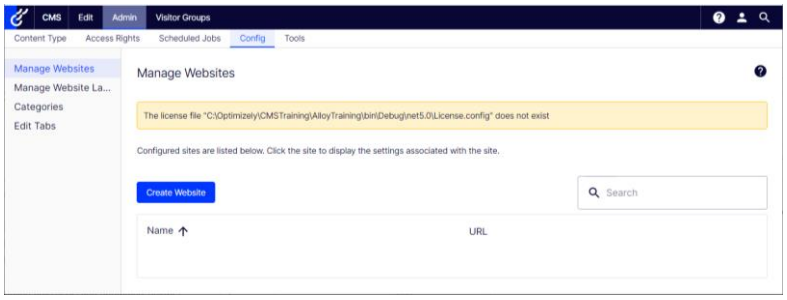
- There is no On-Page Editing (OPE) yet because you have not created a page template.

9. Enter values for **Heading** and **Main body**. Be creative!
    a. Heading: Welcome!
    b. Main body: This is the BEST start page **EVER**!

- Every page has an inherited property named **Category**. If your editors do not want to use it, you can write code to hide it.

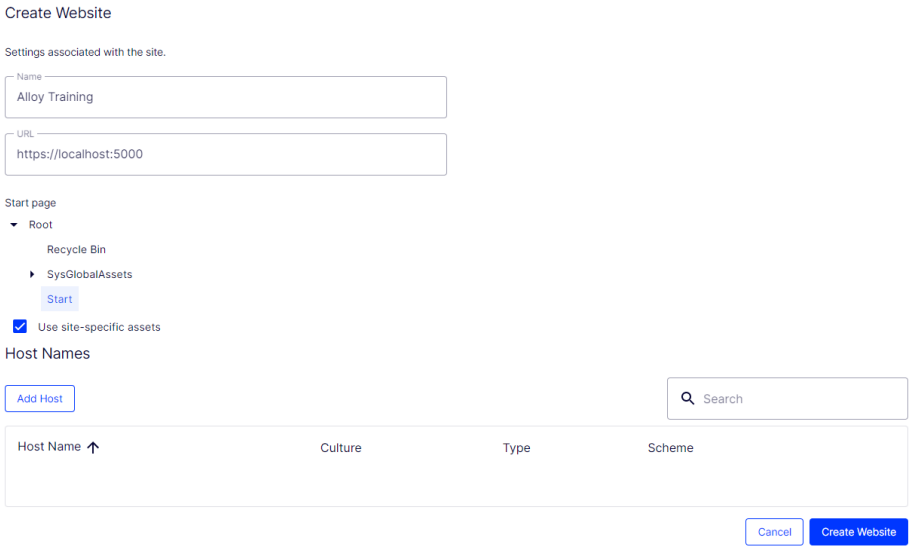
10. Click the **Publish?** button, and then click the **Publish** button.

11. Navigate to **CMS** | **Admin** | **Config** | **Manage Websites**, and click **Create Website**, as shown in the following screenshot:



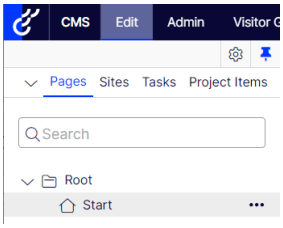
12. Enter the following details, as shown in the following screenshot:
    a. **Name**: Alloy Training
    b. **URL**: `https://localhost:5001`
    c. **Start page**: select the **Start** page you just created.
    d. **Use site-specific assets**: checked

- **Use site-specific assets** enables the **For This Site** folder in the **Assets** pane.



13. Click **Create Website**.

14. Switch to **Edit** view and note that the **Start** page now shows a "house" icon, as shown in the screenshot:



But even if a visitor were to come to the website, the Start page still returns a 404 because it does not yet have a template, so in the next task you will create a page template.

15. Close the browser and shut down the web server.

Creating a Start page template

- A CMS page template in MVC is a combination of a controller and a view.

1. In **AlloyTraining**, create a new folder named **Controllers**.
2. In **Controllers**, right-click and click **Open in Terminal**.
3. At the command line, create a page controller for your previously created StartPage type, as shown in the following command:

```
dotnet new epi-cms-pagecontroller -p:na AlloyTraining.Controllers -ct
StartPage -n StartPageController
```

4. Review the **StartPageController** class, as shown in the following code:

```
using AlloyTraining.Models.Pages;
using EPiServer.Framework.DataAnnotations;
using EPiServer.Web.Mvc;
using Microsoft.AspNetCore.Mvc;

namespace AlloyTraining.Controllers
{
    [TemplateDescriptor(Inherited = true)]
    public class StartPageController : PageController<StartPage>
    {
        public ActionResult Index(StartPage currentPage)
        {
            // Implementation of action. You can create your own view model class that you pass to the
view or
            // you can pass the page type model directly for simpler templates

            return View(currentPage);
        }
    }
}
```

5. In **AlloyTraining**, create a new folder named **Views**.
6. If you are using Visual Studio 2022, then in the **Views** folder, choose **Add** | **New Item...**, or press *Ctrl* + *Shift* + *A*. In **Add New Item - AlloyTraining**, navigate to **Installed** | **Visual C#**, choose **Razor View Imports**, and click **Add**.
7. If you are using Visual Studio Code, then create a new file named **_ViewImports.cshtml**.
8. Import some common `EPiServer` and `Microsoft` namespaces, and enable tag helpers, as shown in the following code:

```
@using EPiServer.Framework.Localization
@using EPiServer.Web.Mvc.Html
@using EPiServer.Shell.Web.Mvc.Html
@using EPiServer.Core
@using EPiServer.Web
```

```
@using EPiServer.Web.Mvc
@using EPiServer.Web.Routing

@using AlloyTraining.Models.Pages

@using Microsoft.AspNetCore.Mvc.Razor
@using Microsoft.AspNetCore.Html

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

9. In **Views**, add a new folder named **StartPage**.
10. In the **StartPage** folder, add a new empty Razor view named **Index.cshtml**.
11. Set the **@model** to use the correct page type, as shown in the following code:

```
@model AlloyTraining.Models.Pages.StartPage
```

12. Add an `<h1>` element to output the **Heading** if it exists, otherwise fall back to the **Name**, as shown in the following markup:

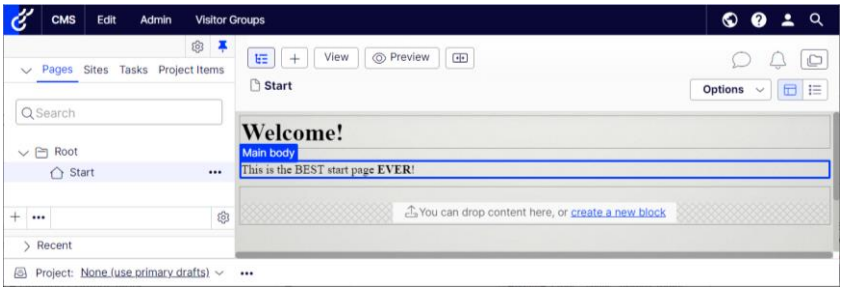
```
<h1 @Html.EditAttributes(m => m.Heading)>
    @(Model.Heading ?? Model.Name)
</h1>
```

13. Add a couple of `<div>` elements to output the main body and the content area properties, as shown in the following markup:

```
<div>
    @Html.PropertyFor(m => m.MainBody)
</div>
<div>
    @Html.PropertyFor(m => m.MainContentArea,
        new { CssClass = "row equal-height" })
</div>
```

14. Start the website, and note the page template is used to render the **Start** page, as shown in the following screenshot:

15. Enter `/episerver/cms/` at the end of the address box, and log in as **Admin**.
16. Navigate to **CMS** | **Edit**., and note the On-Page Editing (OPE) experience due to the page template and use of **PropertyFor** and **EditAttributes** in the view, as shown in the following screenshot:



17. Verify that the on-page edit logic for the `<h1>` tag works, i.e. that when you edit the text it is the **Heading** property value that is updated, and that when a value exists for **Heading** it is that value that is rendered to the visitor, otherwise the **Name** is rendered.
18. Empty the **Heading** property value and publish the change to verify that the fallback works, i.e. that the page name is displayed to the visitor and to the editor.
19. Close the browser and shut down the web server.

Optional: Localizing to the Swedish language for the Start page and its properties

1. Drag and drop or copy the **Resources** folder from **cmsdevfun-exercisefiles\Module B\B1** to the root of the **AlloyTraining** project.

- Make sure you use **AlloyTraining**, because the **AlloyDemo** project already has language files.

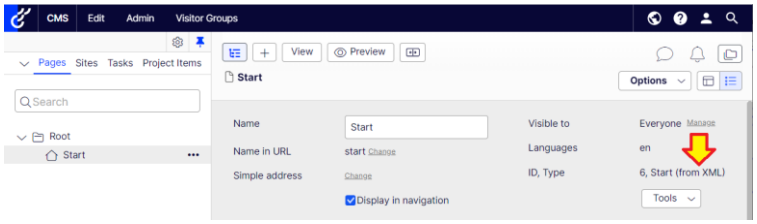
2. Expand the folder named **Resources\Translations** and open the file named **ContentTypes.xml**, and note its contents, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<languages>
  <language name="English" id="en">
    <contenttypes>
      <startpage>
        <name>Start (from XML)</name>
        <description>The home page for a website with an area for blocks and partial
pages.</description>
        <properties>
          <heading>
            <caption>Heading</caption>
            <help>If the Heading is not set, the page falls back to showing the
Name.</help>
          </heading>
          <mainbody>
            <caption>Main body</caption>
            <help>The main body will be shown in the middle of the page, using the
roch text editor you can insert for example text, images and tables.</help>
          </mainbody>
        </properties>
      </startpage>
    </contenttypes>
  </language>
  <language name="Svenska" id="sv">
    <contenttypes>
      <startpage>
        <name>Start</name>
        <description>Hemsidan för en webbplats med ett område för block och
sidor.</description>
        <properties>
          <heading>
            <caption>Rubrik</caption>
            <help>Om rubriken inte är inställd, faller sidan tillbaka för att visa
namnet.</help>
          </heading>
          <mainbody>
            <caption>Huvudkroppen</caption>
            <help>Huvudkroppen kommer att visas i huvudinnehållet på sidan med hjälp
av XHTML-redigeraren som du kan infoga tex, bilder och tabeller.</help>
          </mainbody>
        </properties>
      </startpage>
    </contenttypes>
  </language>
```

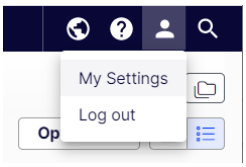
```
</languages>
```

Testing the localization

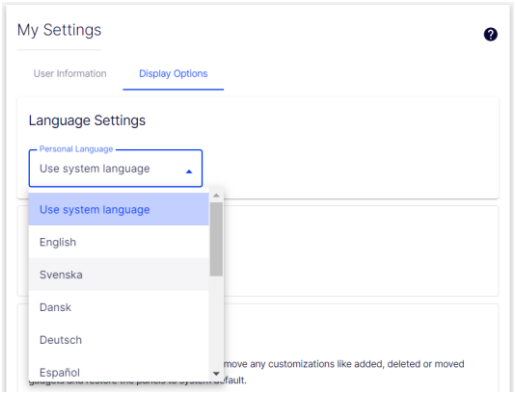
1. Start the **AlloyTraining** website project.
2. Enter **/EPiServer/CMS** at the end of the address box, and log in as Admin.
3. Note that the **Type** of the **Start** page is **Start (from XML)**, which shows that the XML translations are overriding the name set in code, as shown in the following screenshot:



4. In the top menu, click user menu, and then click **My Settings**, as shown in the screenshot:

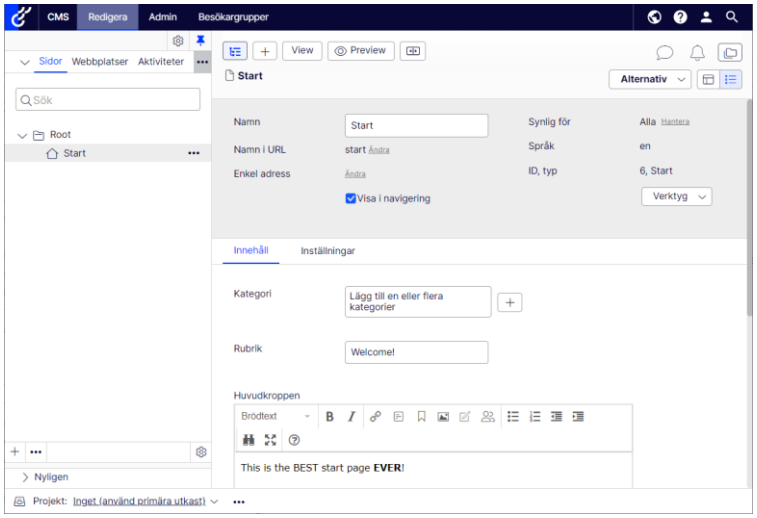


5. Click the **Display Options** tab. The Admin user has their **Personal Language** set to **Use system language**, which on my laptop, is English. Change the language to **Svenska** and then click **Save**, as shown in the following screenshot:

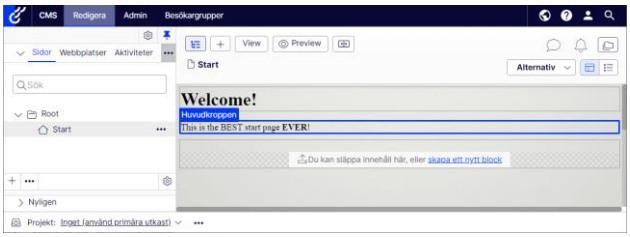


Commented [BG2]: This does not seem to be working.

6. Edit the **Start** page, switch to **All Properties** view, and note the property names and tool tips of your custom **Heading/Rubrik** and **Main body/Huvudkroppen** have been localized into Swedish, as shown in the following screenshot:

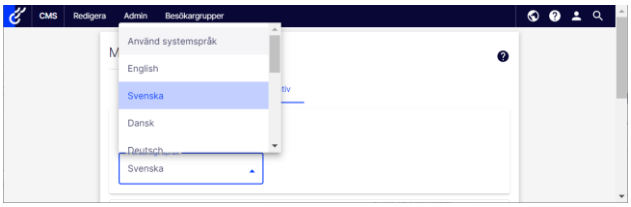


7. Switch to on-page editing and note the same:



- The TinyMCE toolbar hides the overlay if the property has the focus.

8. In the top menu, click the user menu, click **Mina installningar**, click **Visningsalternativ**, and switch back to **Anvand systemsprak**, as shown in the following screenshot:



9. Click **Spara** (Save) and then click **Redigera** (Edit) and note the user interface reverts to English.
10. Close the browser and shut down the web server.

Exercise B2 – Managing media assets

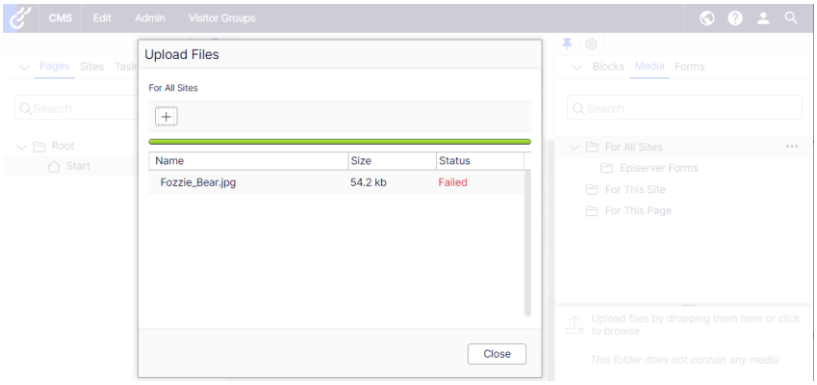
In this exercise, you will define some media types to enable files to be uploaded as CMS assets.

Prerequisites: complete Exercise B1.

Enabling upload of any file by defining a media type

Before uploading asset files, a little code is needed for the system to be able to recognize media. First, let's see the default behavior when an editor tries to upload any file.

1. Start the **AlloyTraining** website project without debugging.
2. Log in as **Admin**.
3. In **Edit** view, in **Assets** pane, click the **Media** tab.
4. Drag and drop any file to the assets pane to try to upload it, but note the **Failed** status, as shown in the following screenshot:



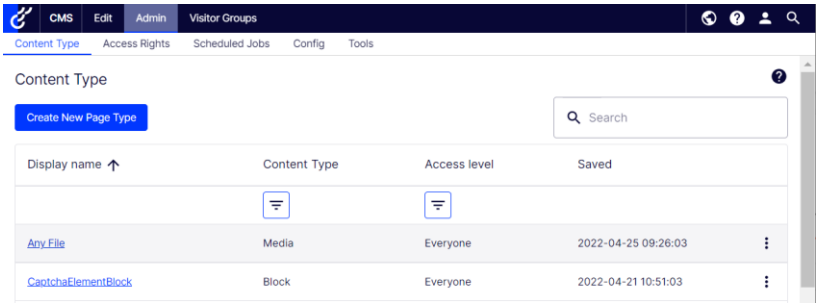
5. Close the browser and shut down the web server.
6. In **Server Explorer**, in **AlloyTraining**, in **Models**, add a folder named **Media**.
7. In the **Media** folder, add a class file named **AnyFile.cs**. We will not bother using the `dotnet new` item template because it does not provide enough benefit.
8. Modify the statements to define a class named `AnyFile` that inherits from `MediaData`, as shown in the following code:

```
using EPiServer.Core; // MediaData
using EPiServer.DataAnnotations; // [ContentType]

namespace AlloyTraining.Models.Media
{
    [ContentType(DisplayName = "Any File",
        Description = "Use this to upload any type of file.")]
    public class AnyFile : MediaData
    {
    }
}
```

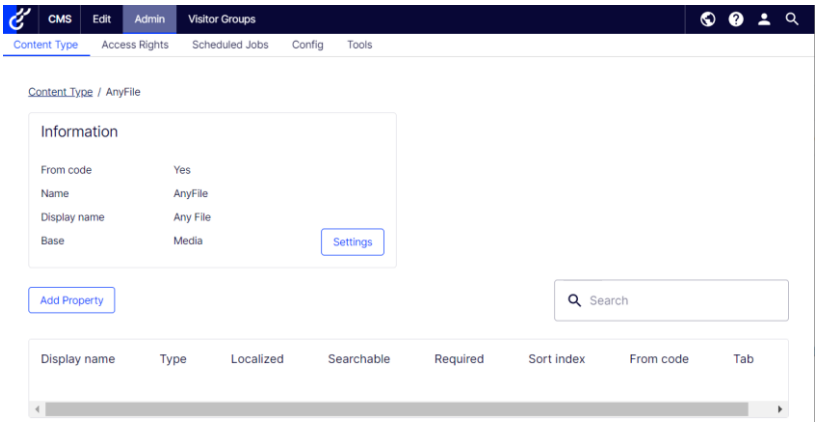
Uploading files

1. Start the **AlloyTraining** website project, and log in as a CMS Admin.

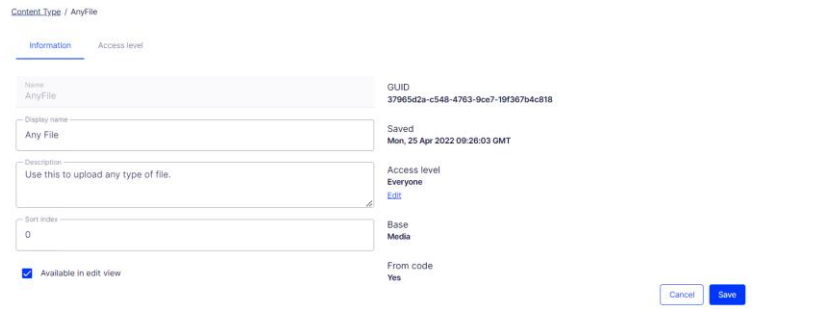
2. Navigate to **CMS** | **Admin** | **Content Type**, and select **Any File**, as shown in the following screenshot:



3. Note AnyFile does not have any custom properties, as shown in the following screenshot:



4. Click the **Settings** button, and note the GUID assigned by the database during the content type synchronization process, as shown in the following screenshot:



5. Copy the GUID value to the clipboard.
6. Close the browser and shut down the web server.
7. In **AnyFile.cs**, add the **GUID** parameter and paste the GUID value, as shown highlighted in the following code:

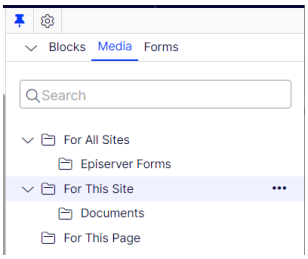
```
[ContentType(DisplayName = "Any File",
    GUID = "37965d2a-c548-4763-9ce7-19f367b4c818",
```

```
    Description = "Use this to upload any type of file.")]
public class AnyFile : MediaData
```

- It is best practice to ensure that the content type class and the content type registration in the CMS database have matching GUIDs so the synchronization process can correctly match them during initialization. If the GUID is missing in the content type class, then it will use the namespace and classname instead. In future, I will not tell you to copy and paste the GUID as shown above but it is recommended good practice. You could also set a GUID in the code first and that value will be used during initial registration in the CMS database.

8. Restart the website project and log in.
9. In the **Assets** pane, in the **For This Site** folder, create a folder named **Documents**, as shown in the following screenshot:

- If you cannot see the **For This Site** folder, then navigate to **CMS** | **Admin** | **Config** | **Manage Websites**, click the **Alloy Training** site, select the **Use site-specific assets** checkbox, and click **Save**.

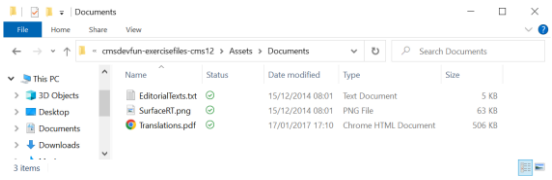


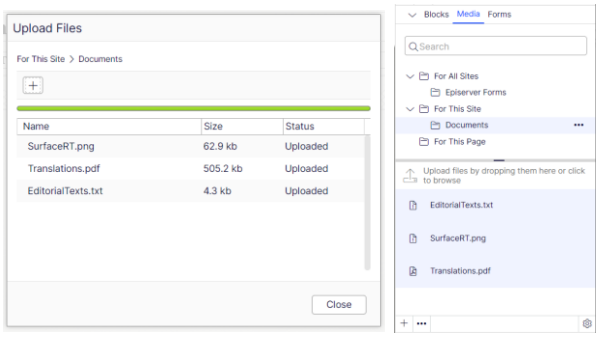
- It does not matter if **Blocks** or **Media** is currently selected when you create a folder because they both share the same folder structure. But you must select the **Media** tab before you can upload files.

10. Make sure that the **Media** tab is selected.

- If the **Blocks** tab is selected, you will not be able to drag and drop files, but it doesn't tell you why.

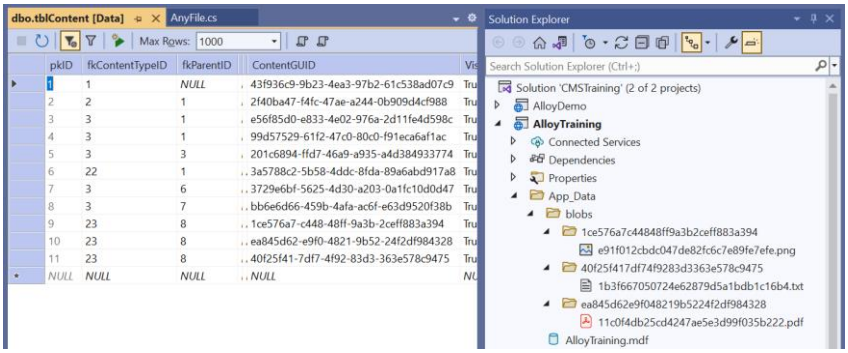
11. Select the **Documents** folder.
12. Start **File Explorer**.
13. From the folder **\cmsdevfun-exercisefiles\Assets\Documents**, drag and drop **SurfaceRT.png**, **EditorialTexts.txt**, and **Translations.pdf** into the **Documents** folder, as shown in the following screenshots:





- If you only *opened* the ZIP instead of *extracted* it, then when you drag and drop the files will not upload. You must drag and drop files that have been extracted from any compressed container.

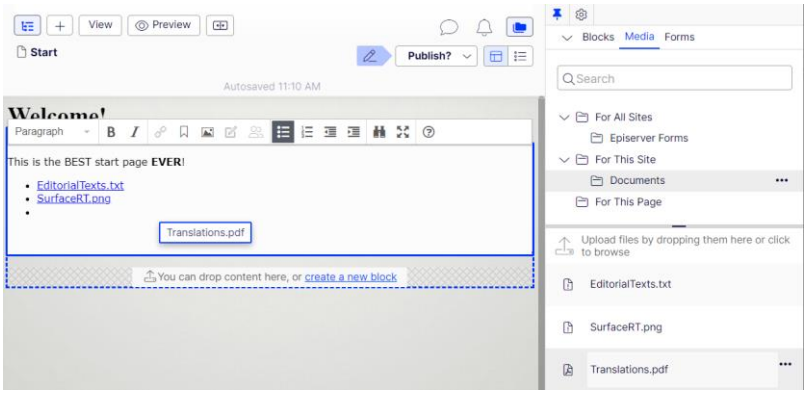
14. Leave the browser running, and in Visual Studio, in **Solution Explorer**, expand the **App_Data** folder, expand the **blobs** folder, and note that each uploaded file has its own folder that matches a content GUID in the CMS database's tblContent table (the three files are in the **Documents** folder that has a **fkParentID** of 8), and a file for a version, as shown in the following screenshot:



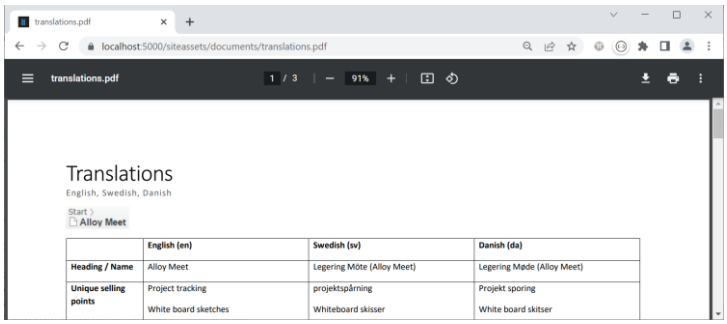
- The default BLOB provider uses the filesystem. You can configure alternative BLOB providers, for example, to use Microsoft Azure Blob Storage or Amazon Web Services S3, and create your own custom providers.

15. Back in the browser, edit the **Start** page.

16. In the **Main body**, create a bulleted list, drag and drop the three files, and note that links to the media files are created, and there is no special handling for images, as shown in the following screenshot:



17. **Publish** the changes to the page.
18. Switch to **Live** view by clicking the globe icon in the blue top menu.
19. Click on each link to see the effect for visitors, as shown in the following screenshot:



4. Close the browser and shut down the web server.

Customize handling of images by defining an image type

Before uploading image files, a little code is needed for the system to be able to recognize images as a special type of media and therefore customize how they are handled when dragged and dropped into a rich text property.

- The previously uploaded image, SurfaceRT.png, will remain registered as an “Any File” content type. To give it the special image handling, it would have to be deleted and uploaded again.

1. In **AlloyTraining**, in **Models**, in **Media**, add a class file named **ImageFile.cs**.
2. Modify the `ImageFile` class to inherit from `ImageData`, have a `DisplayName` of **Image File**, have a `MediaDescriptor` attribute to set the file extensions to: **png,gif,jpg,jpeg**, and have a property to store a text description in the content metadata in the CMS database that can be translated for different cultures, as shown in the following code:

```
using EPiServer.Core; // ImageData
using EPiServer.DataAnnotations; // [ContentType], [CultureSpecific]
using EPiServer.Framework.DataAnnotations; // [MediaDescriptor]
```

```
using System.ComponentModel.DataAnnotations; // [Editable]

namespace AlloyTraining.Models.Media
{
    [ContentType(DisplayName = "Image File",
        Description = "Use this to upload image files.")]
    [MediaDescriptor(ExtensionString = "png,gif,jpg,jpeg")]
    public class ImageFile : ImageData
    {
        [CultureSpecific]
        [Editable(true)]
        public virtual string Description { get; set; }
    }
}
```

- Make sure you inherit from `ImageData`, not `MediaData`! This is what will give it extra capabilities.

Enabling images to be used in a content area

Although content types that derive from `ImageData` have automatic support for drag and drop into a rich text area, to enable them to be dragged and dropped into a content area, they must have a content template. The easiest way to create a media content template is to follow naming conventions:

1. In **AlloyTraining**, in the **Views** folder, add a folder named **Shared**.
2. In the **Shared** folder, add an empty Razor file named **ImageFile.cshtml**.
3. Modify the markup to render the image blob in a figure element with a caption, as shown in the following markup:

```
@model AlloyTraining.Models.Media.ImageFile
<figure>
  <img src="@Url.ContentUrl(Model.ContentLink)"
       style="height:200px; width:200px;"
       alt="@Model.Description" />
  <figcaption>@Model.Name</figcaption>
</figure>
```

Enabling upload of SVG files by defining a media type

`ImageData` does not recognise Scalable Vector Graphics (SVG) files so we must define a separate content type to handle them.

1. In **AlloyTraining**, in **Models**, in **Media**, add a class file named **SvgFile.cs**.
2. Modify the `SvgFile` class to inherit from **ImageData**, change the **DisplayName** to **SVG File**, have a **MediaDescriptor** attribute to set the file extensions to: **svg**, and override the **Thumbnail** property to return the **BinaryData** property, as shown in the following code:

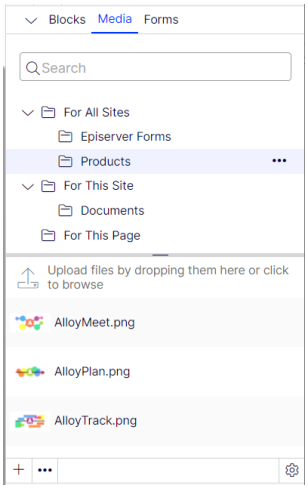
```
using EPiServer.Core;
using EPiServer.DataAnnotations;
using EPiServer.Framework.Blobs;
using EPiServer.Framework.DataAnnotations;

namespace AlloyTraining.Models.Media
{
    [ContentType(DisplayName = "SVG File",
        Description = "Use this to upload Scalable Vector Graphic (SVG)
images.")]
    [MediaDescriptor(ExtensionString = "svg")]
    public class SvgFile : ImageData
    {
        // instead of generating a smaller bitmap file for thumbnail,
        // use the same binary vector image for thumbnail
```

```
            public override Blob Thumbnail { get => base.BinaryData; }
        }
    }
```

- You can read a blog article by Eric Markson about a more detailed handling of SVG files at the following link: https://www.epivisuals.dev/2020/08/RenderingPathXMLFromSvg.html

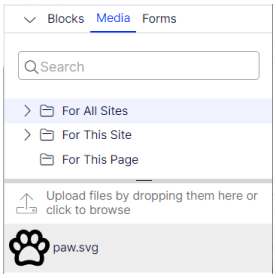
Uploading images

1. Start the **AlloyTraining** site, log in, and navigate to **CMS** | **Edit**.
2. In **Assets** pane, click **Media**, and in **For All Sites**, create a folder named **Products**.
3. Upload images of the three products from **cmsdevfun-exercisefiles.zip** in the folder **\Assets\Products** into the **Products** folder, as shown in the following screenshot:

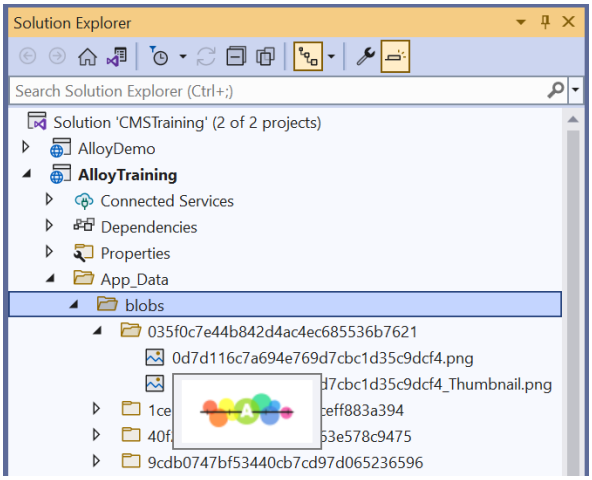


- One feature of inheriting from `ImageData` instead of `MediaData` is the automatic thumbnail generation.

4. Upload the file named **paw.svg** in the folder **\Assets\Misc** into the **For All Sites** folder, as shown in the screenshot:



5. Leave the browser running, and in Visual Studio, in Solution Explorer, in the **App_Data** folder, refresh the **blobs** folder, and note that each uploaded product image has its own folder that matches a content GUID in the CMS database, and a file for a version, and a file for a thumbnail, as shown in the following screenshot:

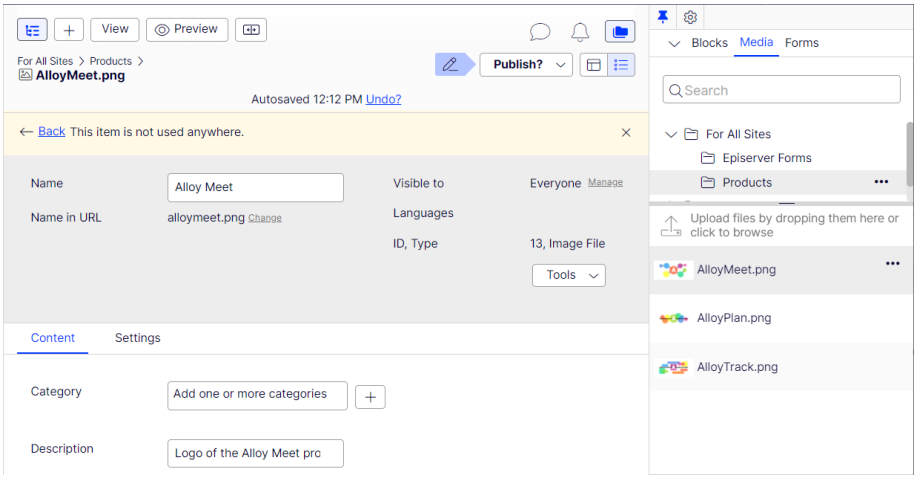


6. Back in the browser, double-click each image in the **Assets** pane **Products** folder, switch to **All Properties** view, edit the **Name** and **Description** properties, and then **Publish** them:

   a. **AlloyMeet.png**:
   Name: **Alloy Meet**, Description: **Logo of the Alloy Meet product**.

   b. **AlloyPlan.png**:
   Name: **Alloy Plan**, Description: **Logo of the Alloy Plan product**.

   c. **AlloyTrack.png**:
   Name: **Alloy Track**, Description: **Logo of the Alloy Track product**.

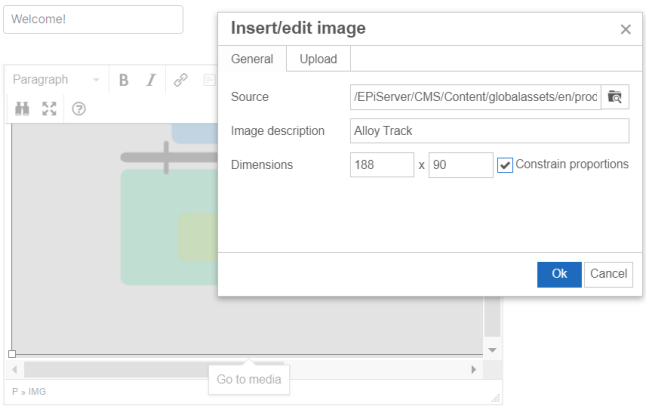


- Make sure that you publish the changes to the image metadata, or in a later exercise you will not see the names and descriptions.

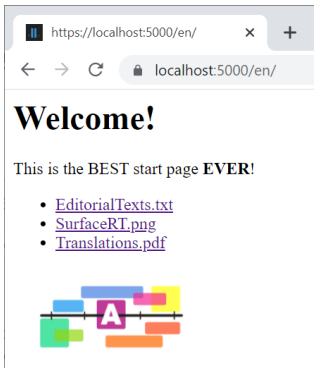
Using image assets

1. Edit the **Start** page.

2. Drag and drop one of the product images into the **Main body** and note the image renders as an `<img>` element instead of a clickable hyperlink.
3. Select the image.
4. In the toolbar, click **Insert/edit image**, and change the width to **188** or the height to **90**, as shown in the screenshot:

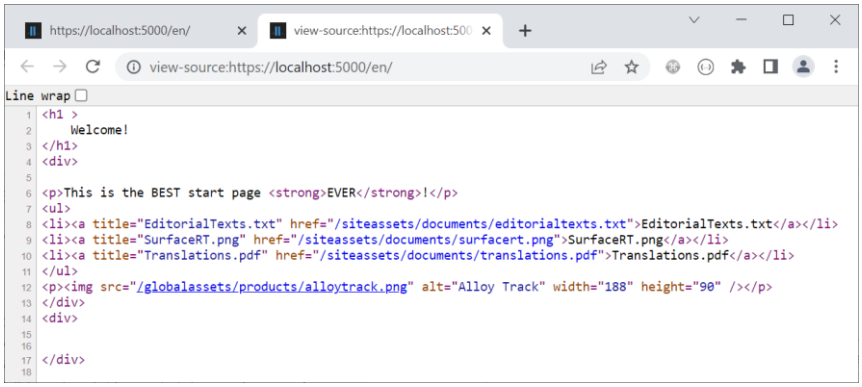


5. Click **OK**.
6. Publish the **Start** page.
7. Switch to **Live view**, to view the page as a visitor, as shown in the following screenshot:



8. Right-click the page and choose **View page source**.

9. Note the HTML generated for visitors, especially the URLs for the documents and images that you uploaded as media assets, as shown in the following screenshot:

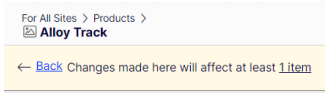


- `/siteassets/` is the **For This Site** folder. `/globalassets/` is the **For All Sites** folder.

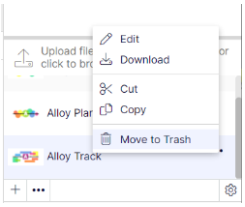
7. Drag and drop an image into the content area, view page source, and note the output includes the figure caption.

Deleting referenced content

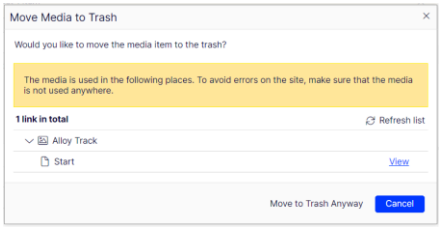
1. Switch to **Edit view**.
2. In **Assets**, on the **Media** tab, select the **For All Sites/Products** folder, double-click the image you dragged and dropped into the rich text area to edit it in **All Properties** view, and note the warning at the top that says it is referenced by one item, as shown in the screenshot:



3. In **Assets**, select the context menu for the image, and choose **Move to Trash**, as shown in the screenshot:



4. Note the warning that this media is used on the Start page, as shown in the following screenshot:



5. Click **Cancel**.

- If you click **Move to Trash Anyway**, then the embedded image on the Start page would return 404. You could then **View Trash** and **Restore** the image back to its original location.

6. Close the browser and shut down the web server.

Handling PDFs in custom ways

`MediaData` does not handle Portable Document Format (PDF) files in any special way, so we might want to define a separate content type to handle them.

- There is an `EPiServer.PdfPreview` 1.0.0 that allows an editor to preview PDF documents in Edit view. The package includes a class named `PdfFile` which handles uploaded files with the .pdf extension. To change this default behavior, a media resolver class named `PdfContentMediaResolver` is used to ignore the `PdfFile` type in the package and thus, the existing registered media type is the candidate: https://docs.developers.optimizely.com/content-cloud/v12.0.0-content-cloud/docs/pdf-preview

4. In **AlloyTraining**, in **Models**, in **Media**, add a class file named **PdfFile.cs**.
5. Modify the `PdfFile` class to inherit from **ImageData**, set the **DisplayName** to **Portable Document Format**, set the **MediaDescriptor** attribute and set the file extensions to: **pdf**, and define a **RenderPreviewImage** property as a boolean.

Your code should look something like the following:

```
using EPiServer.Core;
using EPiServer.DataAnnotations;
using EPiServer.Framework.DataAnnotations;
using System.ComponentModel.DataAnnotations;

namespace AlloyTraining.Models.Media
{
    [ContentType(DisplayName = "Portable Document Format",
        Description = "Use this to upload Portable Document Format (PDF) files.")]
    [MediaDescriptor(ExtensionString = "pdf")]
    public class PdfFile : MediaData
    {
        [Display(Name = "Render preview image")]
        // false: render as simple hyperlink
        // true: render as clickable thumbnail preview image
        public virtual bool RenderPreviewImage { get; set; }
    }
}
```

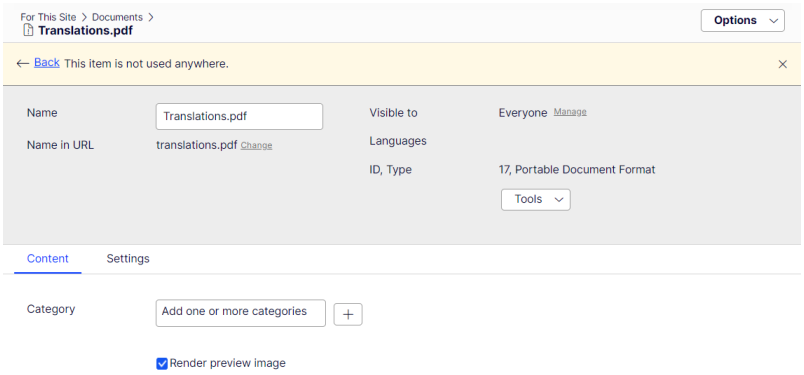
6. In **AlloyTraining**, in the **Views** folder, add a folder named **Shared**.
7. In the **Shared** folder, add an empty Razor file named **PdfFile.cshtml**.
8. Modify the markup to check the `bool` property to change the rendering of a PDF file, as shown in the following markup:

```
@using EPiServer.Web.Mvc.Html
@model AlloyTraining.Models.Media.PdfFile
@if (Model.RenderPreviewImage)
{
    <a href="@Url.ContentUrl(Model.ContentLink)">
        <figure>
            <img src="~/gfx/pdf.png" style="height:50px; width:50px;" />
            <figcaption>@Model.Name</figcaption>
        </figure>
    </a>
}
else
{
    @Html.ContentLink(Model.ContentLink)
}
```

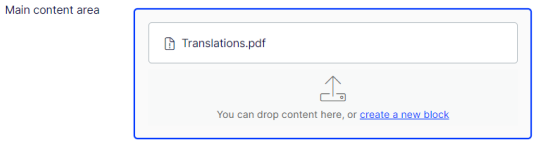
- You would need to have some mechanism to generate the thumbnail image for the PDF, but the template above will give you an idea of how the output can be customized using a property of the PDF content type.

Testing the PDF media type and template

1. Start the **AlloyTraining** site, log in, and navigate to **CMS** | **Edit**.
2. In **Assets** pane, click **Media**, and in the **Documents** folder, move **Translations.pdf** to the Trash.
3. Drag and drop **Translations.pdf** from Windows Explorer into the **Documents** folder to upload it again. This will re-register this media asset with your new PDF media type.
4. Double-click the uploaded file, set its **Render preview image** property, and note it has been properly registered as a PDF content type, as shown in the following screenshot:



5. Publish the change.
6. Edit the **Start** page.
7. Drag and drop the **Translations.pdf** into the **Main content area** property (not the rich text box **Main body**), as shown in the following screenshot:



8. Publish the change.
9. Switch to **Live** view the see the Start page as a visitor, and note the rendered PDF shows the clickable image, as shown in the following screenshot:

10. Close the browser and shut down the web server.

- Note that a PDF file dragged into an `XhtmlString` property like `MainBody` will continue to be rendered as a simple clickable hyperlink, as shown in the bullets in the screenshot. Only PDF files dragged into a `ContentArea` property like `MainContentArea` will use the custom view. Customizing how `XhtmlString` renders content is beyond the scope of this fundamentals course.

Exercise B3 – Implementing design patterns and conventions

- Once you start working on this exercise, do not try to build or run the website until you have completed all the tasks in the exercise. If you build or run the website in the middle of the exercise, then you will see exceptions.

In this exercise, you will create a layout file which will be used up by all page templates in the website, a base page type that will be inherited from by all the page types in the site, a base page controller that will be inherited from by all the page templates in the site, and a view model to make our views and layouts more flexible.

- The layout will be named **_Root.cshtml**.
- The layout will be created in the folder **~\Views\Shared\Layouts** and contain references to Bootstrap files you added in Exercise B1.
- You will use **IContentLoader** to retrieve all pages on the level below the Start page in the page tree.

- The interface and its methods will be explained in more detail later in the course, this exercise merely shows how you can easily retrieve and display pages in the site programmatically.

Prerequisites: complete Exercises B1 and B2.

~\Views\Shared\Layouts_Root.cshtml

```
<head>
</head>
<body>
```

~\Views\StartPage\Index.cshtml

```
</body>
```

Creating a page type base

This class will not be a page type, but it will serve as a base class that inherits and extends `PageData`.

1. In **AlloyTraining**, add a new class file named **SiteTabNames.cs**.
2. Modify the file, as shown in the following code, and note the following:
   - To see properties on the **SEO** tab, the user must have **Edit** access level (i.e. **Change** access right).
   - To see properties on the **Site Settings** tab, the user must have **Administer** access level.

```
using EPiServer.DataAnnotations; // [GroupDefinitions], [RequiredAccess]
using EPiServer.Security; // AccessLevel
using System.ComponentModel.DataAnnotations; // [Display]

namespace AlloyTraining
{
    [GroupDefinitions]
    public static class SiteTabNames
    {
        [Display(Order = 10)] // to sort horizontal tabs
```

```
            [RequiredAccess(AccessLevel.Edit)]
            public const string SEO = "SEO";

            [Display(Order = 20)]
            [RequiredAccess(AccessLevel.Administer)]
            public const string SiteSettings = "Site Settings";
        }
    }
```

3. In **AlloyTraining**, in **Models**, in **Pages**, add a class file named **SitePageData.cs**.
4. Import the `EPiServer.Core`, `EPiServer.DataAbstraction`, `EPiServer.DataAnnotations`, `EPiServer.Web`, and `System.ComponentModel.DataAnnotations` namespaces.
5. Modify the class to be `abstract` and inherit from `PageData`.
6. Define some `public virtual` properties:

- Do not apply any attributes to the properties yet. You will do that in the next section.

    - **MetaTitle**: `string`
    - **MetaKeywords**: `string`
    - **MetaDescription**: `string`
    - **PageImage**: `ContentReference`
7. Apply attributes to the properties to:
    - Make the following properties support a plain text editor with multiple rows: `MetaDescription`.
    - Make `PageImage` property only able to point to images.
    - Make the following properties support multiple language branches: `MetaTitle`, `MetaKeywords`, `MetaDescription`.
    - Make the following properties appear on the `SEO` tab: `MetaTitle`, `MetaKeywords`, `MetaDescription`.
    - Make the following properties appear on the `Content` tab: `PageImage`.
    - Sort the properties within each tab appropriately.
    - Limit the `MetaTitle` to between 5 and 60 characters (Google's recommendation for page titles to get good SEO).

Your complete class should look something like the following:

```
using EPiServer.Core; // PageData, ContentReference
using EPiServer.DataAbstraction; // SystemTabNames
using EPiServer.DataAnnotations; // [CultureSpecific]
using EPiServer.Web; // UIHint
using System.ComponentModel.DataAnnotations; // [Display], [StringLength], [UIHint]

namespace AlloyTraining.Models.Pages
{
    public abstract class SitePageData : PageData
    {
        [CultureSpecific]
        [Display(Name = "Meta title",
            GroupName = SiteTabNames.SEO, Order = 100)]
        [StringLength(60, MinimumLength = 5)]
        public virtual string MetaTitle { get; set; }

        [CultureSpecific]
        [Display(Name = "Meta keywords",
```

```
            GroupName = SiteTabNames.SEO, Order = 200)]
        public virtual string MetaKeywords { get; set; }

        [CultureSpecific]
        [Display(Name = "Meta description",
            GroupName = SiteTabNames.SEO, Order = 300)]
        [UIHint(UIHint.Textarea)] // multi-row text editor
        public virtual string MetaDescription { get; set; }

        [Display(Name = "Page image",
            GroupName = SystemTabNames.Content, Order = 100)]
        [UIHint(UIHint.Image)] // filters to only show images
        public virtual ContentReference PageImage { get; set; }
    }
}
```

8. Drag and drop **\cmsdevfun-exercisefiles\Module B\B3\Resources** folder into **AlloyTraining** project.

- The existing **\Resources\Translations\ContentType.xml** will be overwritten with a new one that has entries to localize any page types that inherit from `SitePageData` and its meta properties. Even without needing Swedish, it is still worth having a localization language file to translate between programmer-speak, e.g. the MetaDescription property, and editor-speak, e.g. “Page description”.

Creating a view model interface

As well as data stored in the page’s content type, every page will share a layout that needs to show the top-level menu, consisting of links to the children of the start page, and a reference to which of those children it is under aka its section. We will define a view model to store this information.

1. In **AlloyTraining**, in **Models**, add a new folder named **ViewModels**.
2. In **ViewModels**, add a new file named **IPageViewModel.cs**.
3. Modify the interface to define a covariant generic interface with a type **T** that must inherit from `SitePageData`, with read-only properties named `CurrentPage`, `StartPage`, `MenuPages`, and `Section`, as shown in the following code:

```
using AlloyTraining.Models.Pages; // SitePageData, StartPage
using EPiServer.Core; // IContent
using System.Collections.Generic; // IEnumerable

namespace AlloyTraining.Models.ViewModels
{
    public interface IPageViewModel<out T> where T : SitePageData
    {
        T CurrentPage { get; }
        StartPage StartPage { get; }
        IEnumerable<SitePageData> MenuPages { get; }
        IContent Section { get; }
    }
}
```

- `Section` property will be used to reference the page that is shown in the top-level navigation menu. For example, the news article **Alloy Saves Bears** is in the **About us** section.

Creating a default view model class

1. In **AlloyTraining**, in **ViewModels**, add a new class file named **PageViewModel.cs**.
2. Modify the class to implement the interface, as shown in the following code:

```
using AlloyTraining.Models.Pages; // SitePageData, StartPage
using EPiServer.Core; // IContent
```

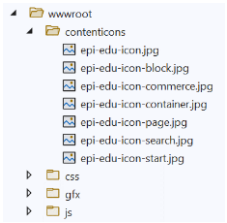
```
using System.Collections.Generic; // IEnumerable

namespace AlloyTraining.Models.ViewModels
{
    public class PageViewModel<T>
        : IPageViewModel<T> where T : SitePageData
    {
        public T CurrentPage { get; set; }
        public StartPage StartPage { get; set; }
        public IEnumerable<SitePageData> MenuPages { get; set; }
        public IContent Section { get; set; }

        public PageViewModel(T currentPage)
        {
            CurrentPage = currentPage;
        }
    }

    public static class PageViewModel
    {
        public static PageViewModel<T> Create<T>(T currentPage)
            where T : SitePageData
        {
            return new PageViewModel<T>(currentPage);
        }
    }
}
```

- The `static PageViewModel` class with its `static Create<T>()` method is a convenience for creating `PageViewModel` instances without having to specify the type because generic methods can use type inference while constructors cannot.

Creating site content icons

If you do not apply the **ImageUrl** attribute, then when editors create new pages and blocks, a missing icon is shown. You will create custom icons and apply them to your content types.

1. In **AlloyTraining**, copy the **contenticons** folder from **\cmsdevfun-exercisefiles\Module B\B3\wwwroot** to the **wwwroot** folder in **AlloyTraining**, as shown in the screenshot:



2. In **AlloyTraining**, add a new class file named **SiteContentIcons.cs**.

- To save time, you could copy **SiteContentIcons.cs** from **\cmsdevfun-exercisefiles\Module B\B3**

3. Modify the contents, as shown in the following code:

```
using EPiServer.DataAnnotations; // ImageUrlAttribute

namespace AlloyTraining
{
    public class SiteImageUrlAttribute : ImageUrlAttribute
    {
        public SiteImageUrlAttribute()
            : base("~/contenticons/epi-edu-icon.jpg") { }

        public SiteImageUrlAttribute(string path)
            : base(path) { }
    }

    public class SitePageIconAttribute : ImageUrlAttribute
```

```
    {
        public SitePageIconAttribute()
            : base("~/contenticons/epi-edu-icon-page.jpg") { }
    }

    public class SiteBlockIconAttribute : ImageUrlAttribute
    {
        public SiteBlockIconAttribute()
            : base("~/contenticons/epi-edu-icon-block.jpg") { }
    }

    public class SiteStartIconAttribute : ImageUrlAttribute
    {
        public SiteStartIconAttribute()
            : base("~/contenticons/epi-edu-icon-start.jpg") { }
    }

    public class SiteSearchIconAttribute : ImageUrlAttribute
    {
        public SiteSearchIconAttribute()
            : base("~/contenticons/epi-edu-icon-search.jpg") { }
    }

    public class SiteCommerceIconAttribute : ImageUrlAttribute
    {
        public SiteCommerceIconAttribute()
            : base("~/contenticons/epi-edu-icon-commerce.jpg") { }
    }

    public class SiteContainerIconAttribute : ImageUrlAttribute
    {
        public SiteContainerIconAttribute()
            : base("~/contenticons/epi-edu-icon-container.jpg") { }
    }
}
```

Modifying Start page to use the base class and icon

1. In **AlloyTraining**, in **Models**, in **Pages**, open **StartPage.cs**.
2. Modify the class to (a) inherit from `SitePageData` and (b) apply the `[SiteStartIcon]` attribute, as shown in the following code:

```
[ContentType(...)]
[SiteStartIcon]
public class StartPage : SitePageData
```

Creating a shared layout

1. In **AlloyTraining**, in **~\Views\Shared**, and add a new folder named **Layouts**.
2. Drag and drop or copy the **_Root.cshtml** file from **\cmsdevfun-exercisefiles\Module B\B3\Views\Shared\Layouts** folder to the same folder in **AlloyTraining**.
3. Open the **_Root.cshtml** file, as shown in the following markup, and note the following:
   - The import of some namespaces for pages, view models and extension methods.
   - `@model` directive allows any type that implements the view model interface to be passed to the layout.
   - The `MetaTitle` (or `Name` as a fallback) of the page is used for the `<title>`
   - Links to stylesheet and script files are added to the `<head>`

- `@await Html.RenderEPiServerQuickNavigatorAsync()` adds the Quick Access menu.

```
@using AlloyTraining.Business.ExtensionMethods
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model IPageViewModel<SitePageData>

<html lang="@Model.CurrentPage.Language">
    <head>
        <meta charset="utf-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=10" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>@(Model.CurrentPage.MetaTitle ?? Model.CurrentPage.Name)</title>
        <meta name="keywords" content="@Model.CurrentPage.MetaKeywords" />
        <meta name="description" content="@Model.CurrentPage.MetaDescription" />
        @Html.CanonicalLink()
        @Html.AlternateLinks()
        <link rel="stylesheet" href="@Url.Content("~/css/bootstrap.css")" />
        <link rel="stylesheet"
              href="@Url.Content("~/css/bootstrap-responsive.css")" />
        <link rel="stylesheet" href="@Url.Content("~/css/media.css")" />
        <link rel="stylesheet" href="@Url.Content("~/css/style.css")" />
        <link rel="stylesheet" href="@Url.Content("~/css/editmode.css")" />
        <script type="text/javascript"
                src="@Url.Content("~/js/jquery.js")"></script>
        <script type="text/javascript"
                src="@Url.Content("~/js/bootstrap.js")"></script>
    </head>
    <body>
        @await Html.RenderEPiServerQuickNavigatorAsync()
        <div class="container">
            <div class="row">
                <div id="header">
                    <div class="span2">
                        <img src="/gfx/logotype.png" />
                    </div>
                    <div class="span10">
                        @if (User.Identity.IsAuthenticated)
                        {
                            <a href="/en/logout">Log out</a>
                        }
                        else
                        {
                            <a href="/util/login/?ReturnUrl=
@Model.CurrentPage.PageLink.ExternalURLFromReference()">Log in</a>
                        }
                    </div>
                </div>
            </div>
            <hr />
            @RenderBody()
        </div>
    </body>
</html>
```

Setting the layout as the default

1. In **AlloyTraining**, in **Views**, add a new **Razor View Start** file named **_ViewStart.cshtml**.
2. Set the layout to the path for **_Root.cshtml**, as shown in the following code:

```
@{
```

```
        Layout = "~/Views/Shared/Layouts/_Root.cshtml";
    }
```

- **_ViewStart.cshtml** is executed when the **View()** method is called inside a controller's action method. It is not executed when the **PartialView()** method is called. This is the only difference between a "full" and "partial" .cshtml file. All .cshtml files themselves are actually the same.

Using the view model in the Start page view

1. Open **~\Views\StartPage\Index.cshtml**.
2. Import namespaces for view models, modify the **@model** directive to use the default view model, and use the **CurrentPage** throughout the view markup, as shown in the following code:

```
@using AlloyTraining.Models.ViewModels
@using EPiServer.Web.Mvc.Html
@model PageViewModel<AlloyTraining.Models.Pages.StartPage>

<h1 @Html.EditAttributes(m => m.CurrentPage.Heading)>
    @(Model.CurrentPage.Heading ?? Model.CurrentPage.Name)
</h1>
<div>
    @Html.PropertyFor(m => m.CurrentPage.MainBody)
</div>
<div>
    @Html.PropertyFor(m => m.CurrentPage.MainContentArea)
</div>
```

Creating a base page controller

To enable every page to have a shared layout with a **Log out** link, and to have access to the content loader service, you will create a base page controller with a **Logout** action method.

1. In **AlloyTraining**, in **Controllers**, add a new class file named **PageControllerBase.cs**.
2. Modify the class to make it `abstract`, restrict the generic type to be derived from `SitePageData`, and to have a `Logout` action method, as shown in the following code:

```
using AlloyTraining.Business.ExtensionMethods; // GetSection extension method
using AlloyTraining.Models.Pages; // SitePageData, StartPage
using AlloyTraining.Models.ViewModels; // IPageViewModel, PageViewModel
using EPiServer; // IContentLoader
using EPiServer.Core; // ContentReference
using EPiServer.Filters; // FilterForVisitor
using EPiServer.ServiceLocation; // Injected
using EPiServer.Shell.Security; // UISignInManager
using EPiServer.Web.Mvc; // PageController
using Microsoft.AspNetCore.Mvc; // IActionResult
using System.Linq; // Cast<T> extension method
using System.Threading.Tasks; // Task<T>

namespace AlloyTraining.Controllers
{
    public abstract class PageControllerBase<T>
        : PageController<T> where T : SitePageData
    {
        protected readonly Injected<UISignInManager> UISignInManager;
        protected readonly IContentLoader loader;

        public PageControllerBase(IContentLoader loader)
```

```
    {
        this.loader = loader;
    }

    public async Task<IActionResult> Logout()
    {
        await UISignInManager.Service.SignOutAsync();
        return RedirectToAction("Index");
    }

    protected IPageViewModel<TPage> CreatePageViewModel<TPage>(
        TPage currentPage) where TPage : SitePageData
    {
        var viewmodel = PageViewModel.Create(currentPage);

        viewmodel.StartPage = loader.Get<StartPage>(ContentReference.StartPage);

        viewmodel.MenuPages = FilterForVisitor.Filter(
            loader.GetChildren<SitePageData>(ContentReference.StartPage))
            .Cast<SitePageData>().Where(page => page.VisibleInMenu);

        viewmodel.Section = currentPage.ContentLink.GetSection();

        return viewmodel;
    }
  }
}
```

- `FilterForVisitor` removes unpublished pages, pages that the current visitor does not have Read access rights to, and pages without a template. You will learn more about this in Module E.

Modifying Start page controller to use the base class

1. In **AlloyTraining**, in the **Controllers** folder, open **StartPageController.cs**.
2. Modify the class to derive from **PageControllerBase**, as shown in the following code:

```
public class StartPageController : PageControllerBase<StartPage>
```

3. Define a constructor that calls the base constructor passing an instance of `IContentLoader`.
4. In the `Index` action method, call the `CreatePageViewModel` method and pass the current page as a parameter, and then pass the view model into the view, as shown in the following code:

```
using AlloyTraining.Models.Pages; // StartPage
using EPiServer; // IContentLoader
using EPiServer.Framework.DataAnnotations; // [TemplateDescriptor]
using Microsoft.AspNetCore.Mvc; // IActionResult

namespace AlloyTraining.Controllers
{
    [TemplateDescriptor(Inherited = true)]
    public class StartPageController : PageControllerBase<StartPage>
    {
        public StartPageController(IContentLoader loader) : base(loader)
        {
        }

        public IActionResult Index(StartPage currentPage)
```

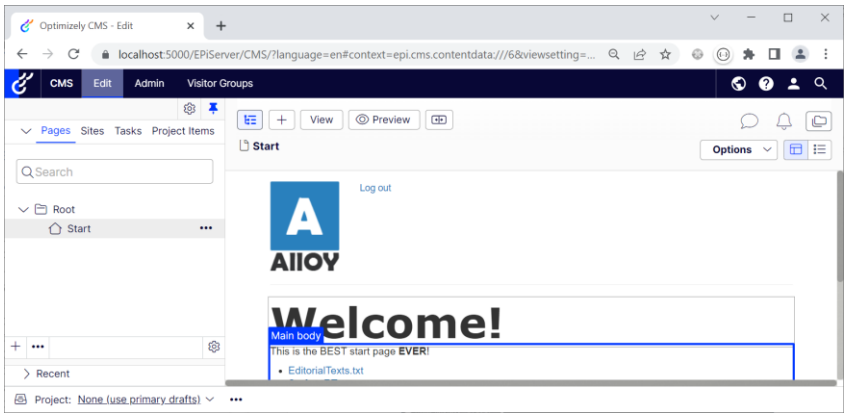
```
            {
                return View(CreatePageViewModel(currentPage));
            }
        }
    }
```

- Calling the inherited `CreatePageViewModel` method creates an instance of the default page view model and sets its current page, section, top level menu, and start page that we will use later.

Testing the website

Finally, we are ready to test the changes.

1. Start the **AlloyTraining** website project and note the shared layout is used for visitors.
2. Log in and note the shared layout is also used for On-Page Editing (OPE), as shown in the following screenshot:



3. Close the browser and shut down the web server.

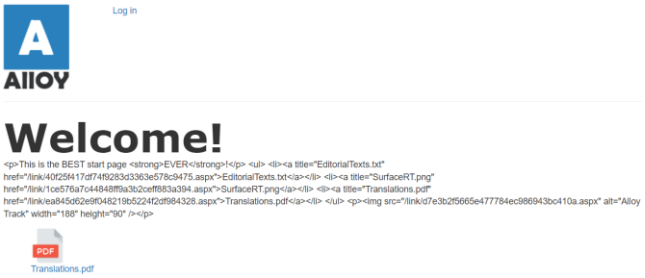
Understanding content GUID links and friendly URLs

It is important to understand what can cause content GUID links to be rendered instead of friendly URLs.

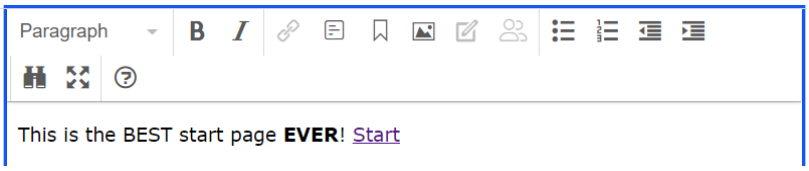
1. In the **AlloyTraining** project, open **Views/StartPage/Index.cshtml**.
2. Modify the rendering of the **MainBody** property so that you aren't using the **PropertyFor()** extension method, as shown in the following markup:

```
<div>
    @Model.CurrentPage.MainBody
</div>
```

3. Start the website, and note the value of the **XhtmlString** property is HTML-encoded so it becomes visible on the page, as shown in the following screenshot:



4. Log in as a CMS Administrator, edit the **Start** page, and switch to **All Properties** view because OPE is disabled when you do not render it with **PropertyFor**.
5. Drag and drop the **Start** page from the **Pages** tree into the **MainBody** to create a link to the page, as shown in the following screenshot:

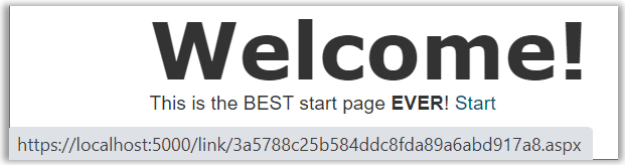


6. Publish the changes to the page.
7. Switch to **Live** view to see the page as a visitor, and note the link to the Start page is stored as a content GUID link, as shown in the following screenshot:

8. Leave the browser running, and in Visual Studio, modify the **Index.cshtml** to wrap the rendering of the **MainBody** property in a call to the **Html.Raw()** method to disable HTML-encoding, as shown in the following markup:

```
<div>
    @Html.Raw(Model.CurrentPage.MainBody)
</div>
```

9. Back in the browser, press *F5* to refresh the page, and note the page seems to be correct, as shown in the following screenshot:



- This is a very common mistake. The content looks correct, and link works, but the use of a content GUID link gives poor SEO and looks ugly.

10. Leave the browser running, and in Visual Studio, modify the **Index.cshtml** to render the **MainBody** property using the **Html.DisplayFor()** method, as shown in the following markup:

```
<div>
    @Html.DisplayFor(m => m.CurrentPage.MainBody)
</div>
```

11. Back in the browser, press *F5* to refresh the page, and note the page looks correct, and it uses friendly URLs for links, as shown in the following screenshot:

- Always remember to render CMS data types using either `DisplayFor()` or `PropertyFor()` to render content hyperlinks using friendly URLs instead of content GUID links.

12. Modify the **Index.cshtml** to render the **MainBody** property using the **Html.PropertyFor()** method to reenable On-Page Editing.
13. Close the browser and shut down the web server.

Defining site settings on Start page for administrators only

It is good practice to store content properties that are shared across the site on the **Start** page for the site. For example, a content reference to a media asset used for the site logo, or some text shown in the footer of the shared layout.

Guidelines for site settings on a start page:

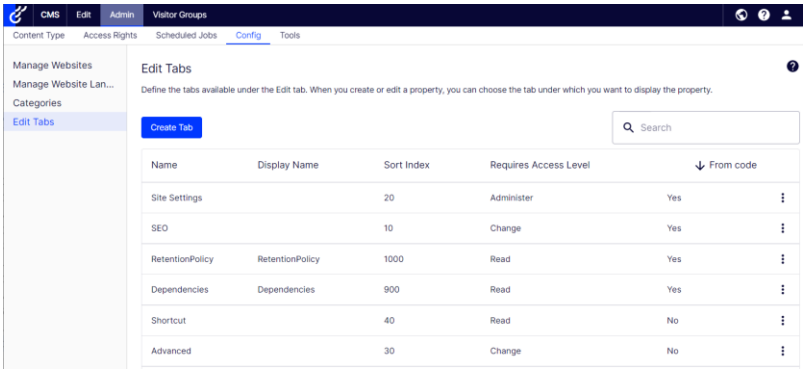
- Must be values that an admin needs to change themselves through the UI.
- Should be protected at the tab level so that non-admins can still change other properties of the Start page.
- Must apply at the site level, not subsections. For subsections, define a page type for a section and store its settings there.
- Must not be developer configuration like paths to APIs. These should be set in **appsettings.json** or environment variables.
- Must never be passwords!

Let's explore an example of a typical site setting.

1. In the **AlloyTraining** project, open **StartPage.cs**.
2. Add a localizable property named **FooterText**, and put it under the **Site Settings** tab, as shown in the following code:

```
[CultureSpecific]
[Display(Name = "Footer text",
    Description = "The footer text will be shown at the bottom of every page.",
    GroupName = SiteTabNames.SiteSettings, Order = 10)]
public virtual string FooterText { get; set; }
```

3. Start the **AlloyTraining** site, and log in as a CMS admin.
4. Navigate to **CMS** | **Admin** | **Config** | **Edit Tabs**, and note that the two tabs defined in **SiteTabNames** static class, **SEO** and **Site Settings**, have been registered from code, as shown in the following screenshot:

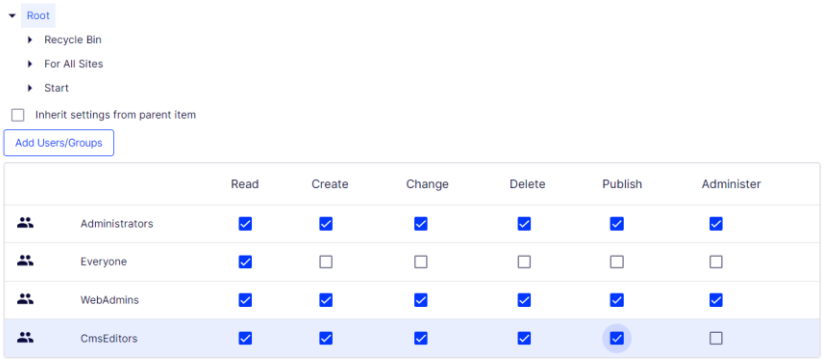


- To see and edit properties on the **SEO** tab, a user must have **Change** access rights. To see and edit properties on the **Site Settings** tab, a user must have **Administer** access rights. Do not allow users to edit a property protected in this way using On-Page Editing because that would bypass this security. Therefore, properties on the **Site Settings** tab should never be output in the view using `PropertyFor`!

5. Navigate to **CMS** | **Admin** | **Access Rights** | **Administer Groups**, and create a new group named **WebEditors**.
6. Navigate to **CMS** | **Admin** | **Access Rights** | **Create User**, and add a new user named **Edward** and make him a member of **WebEditors**.

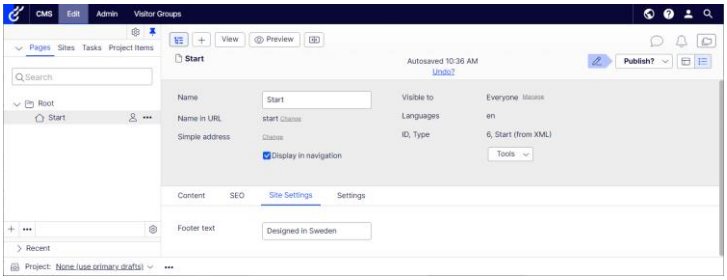
- Remember that in `appsettings.Development.json`, the `CmsEditors` virtual role is mapped to the `WebEditors` stored role/group in whatever authentication provider you have configured. In this case, ASP.NET Core Identity using SQL Server for storage.

7. Navigate to **CMS** | **Admin** | **Access Rights** | **Set Access Rights**, select **Root**, add the virtual role named **CmsEditors**, set all access rights except **Administer**, click **Save Access Rights**, as shown in the following screenshot:



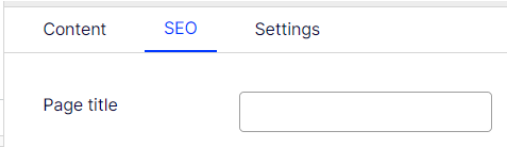
- Since **CmsEditors** do not have **Administer** access rights to the **Root** (and all its children), they will not be able to see or edit properties on the **Site Settings** tab in **All Properties** view.

8. Edit the **Start** page, switch to **All Properties** view, click **Site Settings**, enter a value for the footer text, and note that it has tabs for **SEO** and **Site Settings**, as shown in the following screenshot:



- We are not currently rendering the **Footer text** in the layout. As a challenge, output its value in a `<footer>` element in the **_Root.cshtml** layout. Hint: write an extension method that returns the **FooterText** property from the `StartPage`. A potential solution is in **\cmsdevfun-exercisefiles\Module B\B3\Business\ExtensionMethods\LayoutExtensionMethods.cs** and **\cmsdevfun-exercisefiles\Module B\B3\Views\Shared\Layout_Root-changes.txt**.

9. Publish changes to the page.
10. Log out as a CMS admin, and log back in as **Edward**.
11. Edit the **Start** page, switch to **All Properties** view, and note that **Edward** or other members of **CmsEditors** do not have the ability to see **Site Settings**, as shown in the following screenshot:



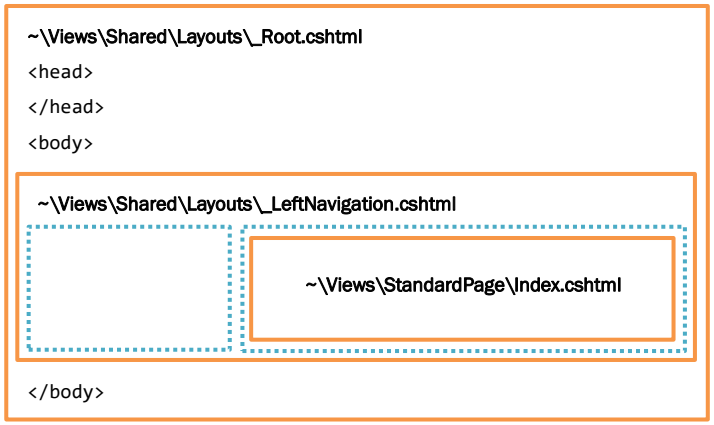
12. Close the browser and shut down the web server.

Exercise B4 – Creating page types with a shared layout and navigation

Prerequisites: complete Exercises B1 to B3.

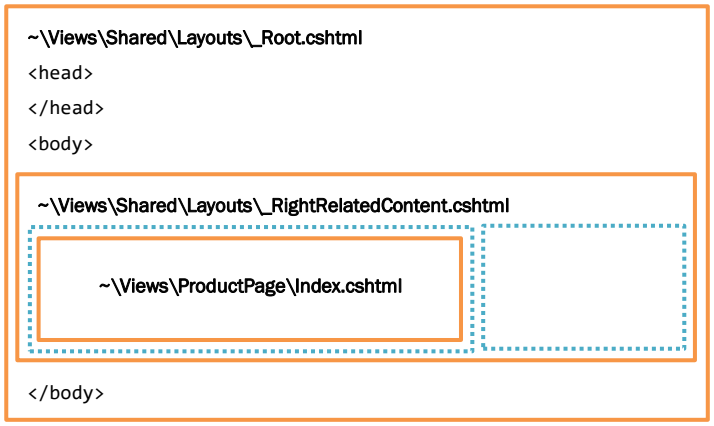
In this exercise, you will create a page type named Standard that will be used for generic pages in the site.

- It will inherit from **SitePageData** class, making sure it gets all the SEO properties and any other properties from this class.
- It will have a **MainBody** property for body text.
- It will use a **_LeftNavigation.cshtml** layout to have a navigation submenu.
- It will have a template, displaying some of the properties on the page type.
- You will create an instance of the **Standard** page named **About us**.



You will create a page type named Product that will be used for product pages in the site.

- It will inherit from **StandardPage**.
- It will have a template with two thirds for main content, and one third for related content.
- You will create three instances of Product page, “Alloy Meet”, “Alloy Track”, and “Alloy Plan”.



You will add a menu to the site to navigate between children of the Start page.

Creating the Standard page type

1. In **AlloyTraining**, in **Models**, in **Pages**, add a new class file named **StandardPage.cs**.
2. Modify the class to inherit from **SitePageData**.
3. Decorate the class with the `ContentType` attribute, change the **DisplayName** to **Standard**, set group to **SiteGroupNames.Common**, and add a **Description** of "Use this page type unless you need a more specialized one."
4. Apply the **[SitePageIcon]** attribute to the class.
5. Add a **MainBody** property with appropriate attribute values.

- The Order in this example is set in relation to the other properties that were added on the **Content** tab in the `SitePageData` base class that this page type inherits from, for example, the `PageImage` property has `Order = 100` and we want the `MainBody` property to be displayed below it in **All Properties** view.

Your code should look something like the following:

```
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;
using System.ComponentModel.DataAnnotations;

namespace AlloyTraining.Models.Pages
{
    [ContentType(DisplayName = "Standard",
        GroupName = SiteGroupNames.Common,
        Description = "Use this page type unless you need a more specialized one.")]
    [SitePageIcon]
    public class StandardPage : SitePageData
    {
        [CultureSpecific]
        [Display(Name = "Main body",
            Description = "The main body will be shown in the main content area of the
page, using the XHTML-editor you can insert for example text, images and tables.",
            GroupName = SystemTabNames.Content,
            Order = 150)]
        public virtual XhtmlString MainBody { get; set; }
    }
}
```

Creating a left navigation layout

1. In **AlloyTraining**, in **Views\Shared\Layouts**, add an empty Razor view named **_LeftNavigation.cshtml**.
2. Modify the view as shown in the following markup, and note the following:
   - `@model` allows any view model that has a `CurrentPage` property whose type derives from `SitePageData` to be passed to the layout.
   - This layout is nested inside the **_Root.cshtml** layout.
   - Bootstrap is used to divide the layout into one third for the (future) left navigation submenu and two thirds for the body of the web page.

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model IPageViewModel<SitePageData>
```

```
@{
    // nest the left navigation inside the root layout
    Layout = "~/Views/Shared/Layouts/_Root.cshtml";
}
<div class="row">
    <div class="span4">
        @RenderSection("_LeftNavigationMenu", required: false)
    </div>
    <div class="span8">
        @RenderBody()
    </div>
</div>
```

Creating a controller for the Standard page type

1. In **AlloyTraining**, in **Controllers**, add a new class named **StandardPageController.cs**.
2. Modify the class to derive from `PageControllerBase<StandardPage>`, and the `Index` action method to use a view model, as shown in the following code:

```
using AlloyTraining.Models.Pages;
using EPiServer;
using Microsoft.AspNetCore.Mvc;

namespace AlloyTraining.Controllers
{
    public class StandardPageController : PageControllerBase<StandardPage>
    {
        public StandardPageController(IContentLoader loader) : base(loader)
        {
        }

        public IActionResult Index(StandardPage currentPage)
        {
            return View(CreatePageViewModel(currentPage));
        }
    }
}
```

- Make sure you pass the view model created by the `CreatePageViewModel` method to the `View` method, not the current page, or you will see a type mismatch runtime exception message.

3. On the **Build** menu, click **Build Solution**.

Creating a view for the Standard page

1. In **AlloyTraining**, in **Views**, add a new folder named **StandardPage**.
2. In the **StandardPage** folder, add an empty Razor file named **Index.cshtml**.
3. Modify the view, as shown in the following markup:

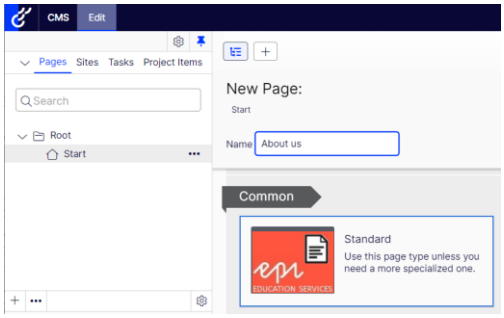
```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<StandardPage>
@{
    Layout = "~/Views/Shared/Layouts/_LeftNavigation.cshtml";
}
<h1 @Html.EditAttributes(m => m.CurrentPage.MetaTitle)>
    @(Model.CurrentPage.MetaTitle ?? Model.CurrentPage.Name)
</h1>
<div class="row">
    <div class="span8 clearfix">
```

```
            @Html.PropertyFor(x => x.CurrentPage.MainBody)
        </div>
    </div>
```

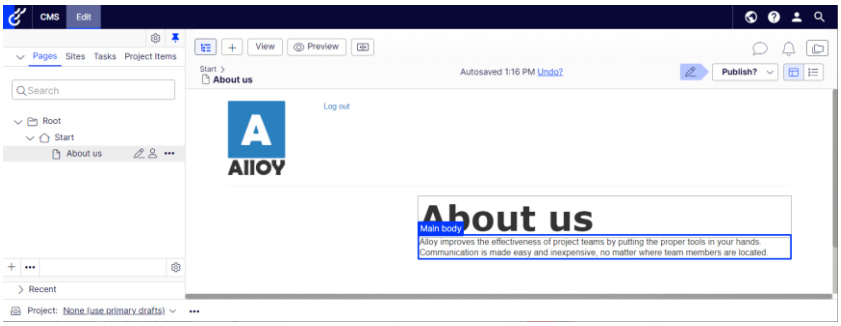
- You would not normally output the `MetaTitle` in the body of a page, but we will do it just as a simple example since we have not defined a `Heading` property.

Creating an instance of the Standard page

1. Start the **AlloyTraining** website and log in.
2. Add a new **Standard** page named **About us** as a child of the **Start** page, as shown in the following screenshot:



3. In on-page editing, set the **Main body** to some text, for example: "Alloy improves the effectiveness of project teams by putting the proper tools in your hands. Communication is made easy and inexpensive, no matter where team members are located."
Note the two-thirds layout with a one-third column on the left ready for some navigation in the future, as shown in the following screenshot:



4. Switch to **All Properties** view and set some approproate **SEO** properties, for example, for the **Page title**: "About us - Alloy Training", and **Page description**: "Alloy improves the effectiveness of project teams by putting the proper tools in your hands. Communication is made easy and inexpensive, no matter where team members are located."
5. Publish the page.
6. Switch to **Live** view and note the page's title.
7. Close the browser and shut down the web server.

Creating a selection factory for themes

A selection factory is used to supply choices for dropdown list boxes and lists of check boxes. The stylesheet defines three color themes that we want a CMS Editor to choose from for a Product page.

1. In **AlloyTraining**, in **Business**, add a new folder names **SelectionFactories**.
2. In **SelectionFactories**, add a new class file named **ThemeSelectionFactory.cs**.
3. Import the `EPiServer.Shell.ObjectEditing` namespace.
4. Modify the class to implement `ISelectionFactory`.
5. Modify the `GetSelections` method to return a list of three select items for stylesheet class themes numbered 1 to 3, as shown in the following code:

```
// ISelectionFactory, ISelectItem, SelectItem, ExtendedMetadata
using EPiServer.Shell.ObjectEditing;
using System.Collections.Generic; // IEnumerable

namespace AlloyTraining.Business.SelectionFactories
{
    public class ThemeSelectionFactory : ISelectionFactory
    {
        public IEnumerable<ISelectItem> GetSelections(ExtendedMetadata metadata)
        {
            yield return new SelectItem { Value = "theme1", Text = "Orange" };
            yield return new SelectItem { Value = "theme2", Text = "Purple" };
            yield return new SelectItem { Value = "theme3", Text = "Green" };
        }
    }
}
```

- In this example you are hard-coding the values for the selection factory. In a real site, you would load and cache the values from an external file or database.

Creating the Product page type

1. In **AlloyTraining**, in **Models**, in **Pages**, add a new class file named **ProductPage.cs**.
2. Modify the class to inherit from **StandardPage**.
3. Change the **DisplayName** to **Product**.
4. Group the page under **Specialized**.
5. Add a **Description** of "Use this for software products that Alloy sells."
6. Apply the **[SiteCommerceIcon]** attribute to the class.
7. Add the following property and decorate it with the theme selection factory so the Editor can choose one theme:
   - Name: `Theme`
   - Type: `string`
8. Set the default theme to **"theme1"**.
9. Add the following property and allow it to be localized into multiple languages:
   - Name: `UniqueSellingPoints`
   - Type: `IList<string>`
   - GroupName: `SystemTabNames.Content`
   - Order: 320
   - Required: force the editor to provide a value when adding a new product page

Your code should look something like the following:

```
using AlloyTraining.Business.SelectionFactories; // ThemeSelectionFactory
using EPiServer.DataAbstraction; // ContentType, SystemTabNames
using EPiServer.DataAnnotations; // [ContentType], [CultureSpecific]
using EPiServer.Shell.ObjectEditing; // [SelectOne]
using System.Collections.Generic; // IList<T>
using System.ComponentModel.DataAnnotations; // [Display], [Required]

namespace AlloyTraining.Models.Pages
{
    [ContentType(DisplayName = "Product",
        GroupName = SiteGroupNames.Specialized, Order = 20,
        Description = "Use this for software products that Alloy sells.")]
    [SiteCommerceIcon]
    public class ProductPage : StandardPage
    {
        public override void SetDefaultValues(ContentType contentType)
        {
            base.SetDefaultValues(contentType);

            Theme = "theme1";
        }

        [SelectOne(SelectionFactoryType = typeof(ThemeSelectionFactory))]
        [Display(Name = "Color theme",
            GroupName = SystemTabNames.Content, Order = 310)]
        public virtual string Theme { get; set; }

        [CultureSpecific]
        [Display(Name = "Unique selling points",
            GroupName = SystemTabNames.Content, Order = 320)]
        [Required]
        public virtual IList<string> UniqueSellingPoints { get; set; }
    }
}
```

Creating a right related content layout

1. In **Views\Shared\Layouts**, add a new empty Razor file named **_RightRelatedContent.cshtml**.
2. Modify the view as shown in the following markup, and note the following:
   - `@model` allows any view model that has a **CurrentPage** property that derives from `SitePageData` to be passed to the layout.
   - This layout is nested inside the **_Root.cshtml** layout.
   - Bootstrap is used to divide the layout into two thirds for the body of the web page and one third for the (optional) related content.

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model IPageViewModel<SitePageData>
@{
    // nest the right related content inside the root layout
    Layout = "~/Views/Shared/Layouts/_Root.cshtml";
}
<div class="row">
    <div class="span8">
        @RenderBody()
    </div>
    <div class="span4">
```

```
            @RenderSection("RelatedContent", required: false)
        </div>
    </div>
```

Creating a controller for the Product page type

1. In **AlloyTraining**, in **Controllers**, add a new class file named **ProductPageController.cs**.
2. Modify the class to derive from **PageControllerBase<T>**, and the **Index** action method to use a view model, as shown in the following code:

```
using AlloyTraining.Models.Pages;
using EPiServer;
using Microsoft.AspNetCore.Mvc;

namespace AlloyTraining.Controllers
{
    public class ProductPageController
        : PageControllerBase<ProductPage>
    {
        public ProductPageController(IContentLoader loader) : base(loader)
        {
        }

        public ActionResult Index(ProductPage currentPage)
        {
            return View(CreatePageViewModel(currentPage));
        }
    }
}
```

3. On the **Build** menu, click **Build Solution**.

Creating a view for the Product page

1. In **AlloyTraining**, in **Views**, add a new folder named **ProductPage**.
2. In the **ProductPage** folder, add a new empty Razor view named **Index.cshtml**.
3. Modify the view as shown in the following mark, noting the following:
   - This view is nested inside the **_RightRelatedContent.cshtml** layout.
   - The body of the view will be injected into the layout where **RenderBody()** was called, inside the first two-thirds .
   - The section **RelatedContent** will be injected into the layout inside the last third .

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<ProductPage>
@{
    Layout = "~/Views/Shared/Layouts/_RightRelatedContent.cshtml";
}
<h1 @Html.EditAttributes(x => x.CurrentPage.Name)>@Model.CurrentPage.Name</h1>
<p class="introduction"
   @Html.EditAttributes(x => x.CurrentPage.MetaDescription)>
   @Model.CurrentPage.MetaDescription</p>
<div class="row">
    <div class="span8 clearfix">
        @Html.PropertyFor(x => x.CurrentPage.MainBody)
    </div>
</div>
@section RelatedContent
```

```
{
    <div @Html.EditAttributes(x => x.CurrentPage.PageImage)>
        <img src="@Url.ContentUrl(Model.CurrentPage.PageImage)" />
    </div>
    <div class="block colorBox @Model.CurrentPage.Theme">
        <h2 @Html.EditAttributes(x => x.CurrentPage.Name)>
            @Model.CurrentPage.Name
        </h2>
        <p @Html.EditAttributes(x => x.CurrentPage.UniqueSellingPoints)>
            @foreach(string usp in Model.CurrentPage.UniqueSellingPoints)
            {
                <span class="label label-inverse">@usp</span>
            }
        </p>
    </div>
}
```

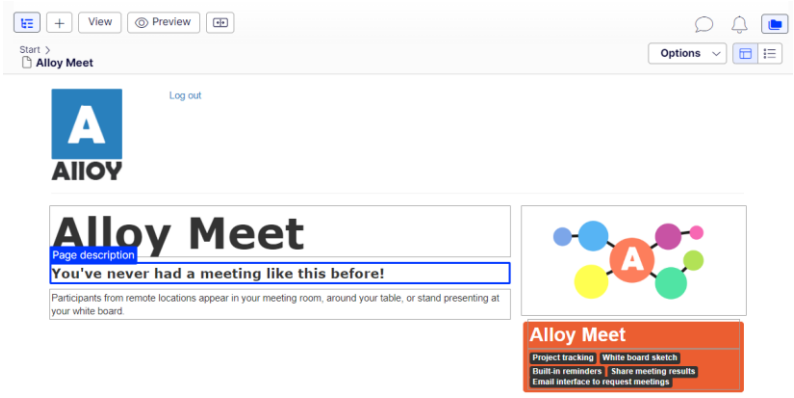
Creating instances of Product page type

- Instead of following these steps to manually create the three product pages, you can drag and drop **\cmsdevfun-exercisefiles\Module B\B4\Business\Initializers\AddPagesInitializer.cs** into your project and in **Startup.cs**, in **ConfigureServices**, import namespaces to enable the `TryAddEnumerable` extension method and an interface to register a dependency service that executes only on the first request. Then start the website which will create the **About us** page and the three product pages using code. You must have created a **Products** folder in **For All Sites** and uploaded the product images into it.

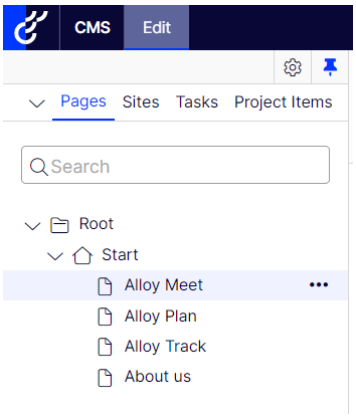
1. Build and start the **AlloyTraining** website.
2. Add three **Product** pages under the **Start** page, using the following table:

	Alloy Meet	Alloy Plan	Alloy Track
Unique selling points	• Project tracking • White board sketch • Built-in reminders • Share meeting results • Email interface to request meetings	• Project planning • Reporting and statistics • Email handling of tasks • Risk calculations • Direct communication to members	• Shared timeline • Project emails • To-do lists • Workflows • Status reports
Page description	You've never had a meeting like this before!	Project management has never been easier!	Projects have a natural lifecycle with well-defined stages.
Main body	Participants from remote locations appear in your meeting room, around your table, or stand presenting at your white board.	Planning is crucial to the success of any project. Alloy Plan takes into consideration all aspects of project planning; from well-defined objectives to staffing, capital investments and management support. Nothing is left to chance.	From start-up meetings to final sign-off, we have the solutions for today's market-driven needs. Leverage your assets to the fullest through the combination of Alloy Plan, Alloy Meet and Alloy Track.
Color theme	Orange	Purple	Green

3. Set **PageImage** for each product page to the appropriate media asset, and publish the pages, as shown in the following screenshot:



4. In **Navigation** pane, on **Pages** tab, drag and drop the product pages to manully order them alphabetically, with **About us** last, as shown in the following screenshot:



- The default sort order for child pages is by creation date with most recent at the top. Although we could set the **Start** page to sort its children alphabetically, that would put **About us** first. By dragging any of its child pages, the **Start** page's sort order will be changed to use the **Index** of each child page. The code that creates the pages for you sets the Index to 100, 200, 300 and 400 to sort them.

Limiting available page types

To improve the experience for Editors, you should limit their choices of page type at points within the page tree structure. In this task, you will prevent the Start page from being created as a child of itself and prevent Product pages under anything other than a Start page.

1. In **AlloyTraining**, in **Models/Pages**, open **StartPage.cs**.
2. Modify the start page type to limit its children to standard pages or their derived page types, as shown highlighted in the following code:

```
[ContentType(...)]
[SiteStartIcon]
```

```
[AvailableContentTypes(Include = new[] { typeof(StandardPage) })]
public class StartPage : SitePageData
```

- By default, all page types are allowed as children under a page type. When you decorate a page type with the [AvailableContentTypes] attribute and explicitly set an array of page types to include (i.e. allow as children), all other page types are *implicitly excluded*.

3. In **AlloyTraining**, open **StandardPage.cs**.
4. Modify the standard page type to limit its children to standard pages or their derived page types, except product pages, as shown highlighted in the following code:

```
[ContentType(...)]
[SitePageIcon]
[AvailableContentTypes(Include = new[] { typeof(StandardPage) },
    Exclude = new[] { typeof(ProductPage) })]
public class StandardPage : SitePageData
```

- If you were to now try to create a **Start** page under the existing **Start** page, it will not be shown in the list of choices, but you could create one under the **Root**. If you were to now try to create a **Product** page under the **About us** page, it will not be shown in the list of choices, but you could create one under **Start**.

Creating a partial view for the navigation menu

Shared website components like navigation menus can be created as partial views for easier reuse.

1. In **AlloyTraining**, in **Views\Shared**, add a new empty Razor file named **_NavigationMenu.cshtml**.
2. Modify the view, as shown in the following markup:

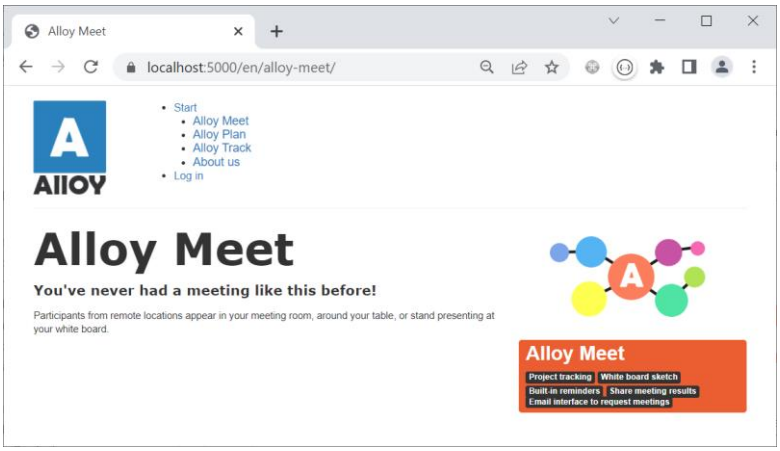
```
@using AlloyTraining.Business.ExtensionMethods
@using AlloyTraining.Models.Pages
@using AlloyTraining.Models.ViewModels
@using EPiServer.Core
@using EPiServer.Web.Mvc.Html

@model IPageViewModel<SitePageData>

<ul>
    <li>
        @Html.ContentLink(ContentReference.StartPage)
        <ul>
            @foreach (SitePageData page in Model.MenuPages)
            {
                <li>@Html.ContentLink(page.ContentLink)</li>
            }
        </ul>
    </li>
    <li>
        @if (User.Identity.IsAuthenticated)
        {
            <a href="/en/logout">Log out @User.Identity.Name</a>
        }
        else
        {
            <a href="/util/login?ReturnUrl=
            @Model.CurrentPage.PageLink.ExternalURLFromReference()">Log in</a>
        }
    </li>
</ul>
```

Updating the root layout to use the navigation menu

1. Open **~\Views\Shared\Layouts_Root.cshtml**.
2. Inside the `<div class="span10">`, delete the `@if` statement that outputs the log in/out hyperlinks, and replace it with a call to output the navigation menu partial view, as shown in the following markup:

```
<div class="span10">
    @await Html.PartialAsync("_NavigationMenu")
</div>
```

3. Start the **AlloyTraining** website, and use the menu to navigate between pages, as shown in the following screenshot:



Helping editors by revealing help text

We have been setting a **Description** for most properties, but you probably haven't seen where they appear because the default behaviour is quite hidden. A CMS Editor must know to hover their mouse over the label for a property and wait a few seconds for a tooltip to appear.

This works because CMS reads the Description of a property and sets it as the `title` attribute on the label which browsers then show as a tooltip, for example:

```
<label title="Between 5 and 60 characters that will be shown in Google search
results.">Page title</label>
```

We will make it more obvious by using CSS to copy the title attribute onto an information icon.

1. In the **AlloyTraining** project, add a new folder named **ClientResources**.
2. In the **ClientResources** folder, add a new folder named **Styles**.
3. In the **Styles** folder, copy the CSS stylesheet named **help-text-icon.css** from the **\exercisefiles\Module B\B4\ClientResources\Styles** folder.
4. Review its contents, as shown in the following code:

```
.Sleek .dijitTabPaneWrapper .epi-form-container__section__row label[title]:after {
    background-color: #C9C9C9;
    border-radius: 10px 10px 10px 10px;
    border: 1px solid #ACACAC;
    color: #FFFFFF;
    content: "i";
    display: inline-block;
    font-size: 1em;
```

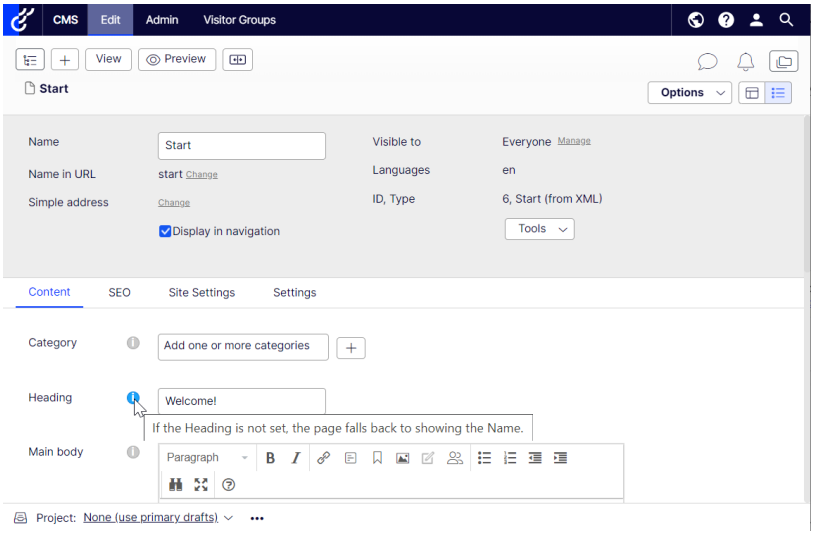
```
    font-weight: bold;
    height: 14px;
    line-height: 14px;
    margin: 0 0 0 5px;
    float: right;
    text-align: center;
    width: 14px;
}
.Sleek .dijitTabPaneWrapper .epi-formsRow label[title=""]:after,
.Sleek .dijitTabPaneWrapper .epi-form-container__section__row label[title=""]:after {
    display: none;
}
.Sleek .dijitTabPaneWrapper .epi-formsRow label[title]:hover:after,
.Sleek .dijitTabPaneWrapper .epi-form-container__section__row label[title]:hover:after
{
    background-color: #1ba4fa;
    border: 1px solid #1285de;
}
```

5. In the **AlloyTraining** project, in the root of the project, add a new item, select **Data** | **XML File**, and name the file **module.config**.
6. Modify its contents, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<module>
    <clientResources>
        <add name="epi-cms.widgets.base"
             path="Styles/help-text-icon.css" resourceType="Style"/>
    </clientResources>
</module>
```

- By naming the client resource **epi-cms.widgets.base**, the stylesheet will modify the editing experience in All Properties view.

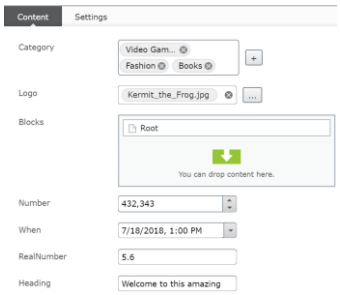
7. Start the **AlloyTraining** website and log in as a CMS administrator.
8. Edit the **Start** page and switch to **All Properties** view.
9. Hover your mouse over any of the "information" icons and note they change to blue and show a tooltip, as shown in the following screenshot:



- Read more about how this technique works on Daved Artemik's blog article: https://beendaved.blogspot.co.uk/2016/09/simple-approach-to-tooltip-icons-for.html

Widening property editors in All Properties view

By default, some editors for common property types are too narrow, as shown in the screenshot:




```
public virtual string Heading { get; set; }
public virtual int Number { get; set; }
public virtual DateTime When { get; set; }
public virtual double RealNumber { get; set; }
public virtual ContentArea Blocks { get; set; }
[UIHint(UIHint.Image)]
public virtual ContentReference Logo { get; set; }
```

In this task, we will use CSS to widen the editors for common property data types.

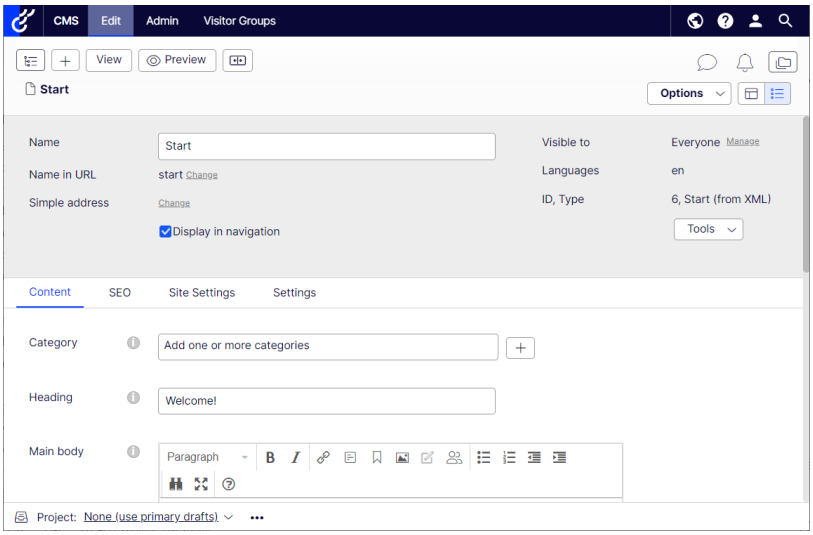
1. In the **AlloyTraining** project, in the **ClientResources\Styles** folder, add a new CSS stylesheet named **widen-editors.css**.
2. Modify its contents, as shown in the following code:

```
/* Widen inputs in All Properties view. */

/* Affects: string, int, double, DateTime, ContentReference, CategoryList */
.Sleek .dijitTextBox {
    width: 425px !important;
}

/* Affects: ContentArea, Image thumbnail */
.Sleek .epi-content-area-editor, .Sleek .thumbnail-editor {
    width: 550px !important;
}
```

3. In the root of the project, open and modify **module.config**, to add a new entry to reference the CSS stylesheet you just created, as shown in highlighted in the following markup:

```
<module>
  <clientResources>
    <!-- paths are relative to /ClientResources/ -->
    <add name="epi-cms.widgets.base"
         path="Styles/help-text-icon.css" resourceType="Style"/>
    <add name="epi-cms.widgets.base"
         path="Styles/widen-editors.css" resourceType="Style" />
  </clientResources>
</module>
```

4. Start the **AlloyTraining** website and log in as a CMS administrator.
5. Edit the **Start** page and switch to **All Properties** view, and note the wider editors, including for the Name in the grey Basic Information Area, as shown in the following screenshot:



6. Close the browser and shut down the web server.

- Due to caching, changes the CSS stylesheet may not be shown. Press *Ctrl + F5* in your browser to force a non-cached request to refresh the stylesheet.

7. Open and modify **module.config** to remove the entry that widens the editors. This is because in a later exercise I want you to see the normal widths.
8. Save changes.

- Another good reason to remove the entry in **module.config** that widens the editors is that it has not been tested in all scenarios and might affect parts of the user interface that you don't expect.

Backing up the AlloyTraining project

1. In **Solution Explorer**, right-click the solution and choose **Open Folder in File Explorer**.
2. In Visual Studio, save all files and then close **Visual Studio** to release any file locks.
3. In **File Explorer**, copy and paste the **AlloyTraining** project folder. You can now use this copied folder as a backup in case you want to complete a later exercise with a working website project up to the end of Exercise B4, for example, exercises D5 onwards.

Module C – Rendering Content Templates

Goal

The overall goal of the exercises in this module is to see how you can customize the experience for visitors. You will:

1. Implement a content area property on the Start page and create a partial page template for Product pages to enable them to be rendered in the content area.
2. Create a partial template for all pages to enable them to be rendered in a content area.
3. Define display options to allow content editors to apply tags to select between multiple templates.
4. Apply tags programmatically to allow developers to apply tags to select between multiple templates
5. Create a display channel to set a tag automatically based on information in an incoming request to select between multiple templates.

Exercise C1 – Creating partial templates for product pages and image files

In this exercise, you will create a content area on the start page and add pages (and later blocks) to it.

Prerequisites: complete Exercises B1 to B4.

Adding a content area to the Start page type and template

The content area on the Start page will allow only blocks or standard pages to be included.

1. In **AlloyTraining**, open **~\Models\Pages\StartPage.cs**.
2. Import the following namespace:

   ```
   using AlloyTraining.Models.Media;
   ```

3. Decorate the **ContentArea** property named **MainContentArea** with appropriate attributes, as shown in the following code:

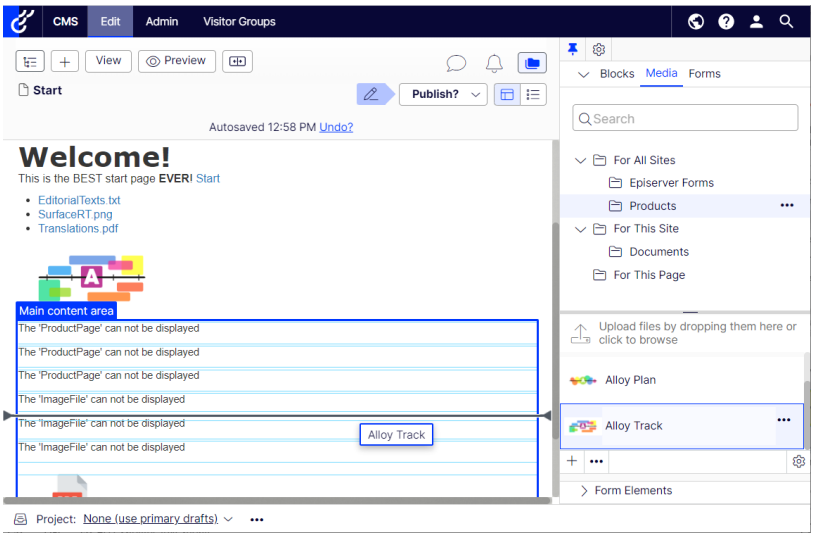
- **MainContentArea** must only allow content references to: Standard pages, blocks, images, PDF files, and folders.

```
[CultureSpecific]
[Display(Name = "Main content area",
    Description = "Drag and drop images, blocks, folders, and pages with
partial templates.",
    GroupName = SystemTabNames.Content,
    Order = 30)]
[AllowedTypes(typeof(StandardPage), typeof(BlockData),
    typeof(ImageData), typeof(ContentFolder), typeof(PdfFile))]
public virtual ContentArea MainContentArea { get; set; }
```

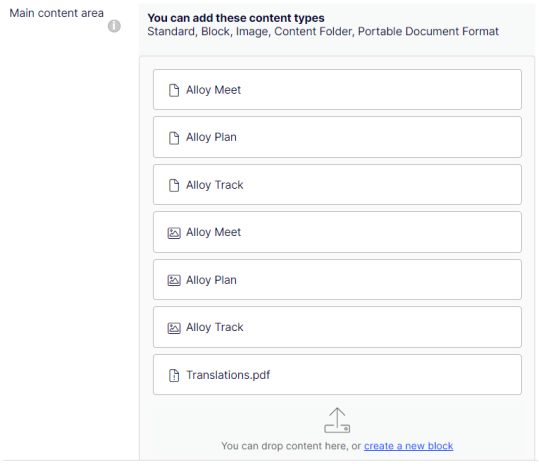
Testing the content area

1. Start the **AlloyTraining** website, and log in as a CMS admin.

2. In **Edit** view, open **Navigation** | **Pages**, drag and drop the three product pages into the **Main content area**, and note the warning message "The 'ProductPage' cannot be displayed".
3. Open **Assets** | **Media**, drag and drop the three product images from the **Products** folder into the **Main content area**, and note the warning message, "The 'ImageFile' cannot be displayed.", as shown in the following screenshot:



4. In **All Properties** view, **Main content area** looks like the following screenshot:



5. **Publish** the changes to the **Start** page.

- If you try to drag and drop the **Start** page into the content area it is not allowed because the **MainContentArea** only allows **StandardPage**, **BlockData**, **ContentFolder**, **ImageData**, or **PdfFile** content.

6. Switch to visitor view. Note that instead of showing the warning messages, visitors see nothing for those unsupported content types because they are automatically skipped over.

- You will fix this issue by creating a partial page template for product pages and a partial template for images.

Creating a partial content controller for product pages

1. In **AlloyTraining**, add a new folder named **Components**.
2. In **Components**, add a new class file named **ProductPagePartialComponent.cs**.
3. In **ProductPagePartialComponent.cs**, import the `EPiServer.Web.Mvc` namespace, and change the class to inherit from `PartialContentComponent<ProductPage>`.
4. Implement the abstract class and its `InvokeComponent` method to use `View` and call `PageViewModel`'s `Create` method passing the current page, as shown in the following code:

```
using AlloyTraining.Models.Pages; // ProductPage
using AlloyTraining.Models.ViewModels; // PageViewModel
using EPiServer.Web.Mvc; // PartialContentComponent
using Microsoft.AspNetCore.Mvc; // IViewComponentResult

namespace AlloyTraining.Components
{
    public class ProductPagePartialComponent :
        PartialContentComponent<ProductPage>
    {
        protected override IViewComponentResult InvokeComponent(
            ProductPage currentPage)
        {
            return View(PageViewModel.Create(currentPage));
        }
    }
}
```

Creating a partial view for product pages

1. In **AlloyTraining**, in **Views\Shared**, add a new folder named **Components**.
2. In the **Components** folder, add a new folder named **ProductPage**.
3. In the **ProductPage** folder, create a Razor file named **Default.cshtml**.
4. Modify the contents, as shown in the following markup, and note the following:
   - The markup is like the "full" view, but it outputs only four properties: **Name**, **MetaDescription**, **UniqueSellingPoints**, and **PageImage**. This is a subset of the content.
   - The property output is wrapped in a with a border, wrapped in a with a Bootstrap **span4** class. This will allocate one third of a Bootstrap **row** width to each product.
   - The whole product is wrapped in a clickable hyperlink that would navigate the visitor to the "full" product page.

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<ProductPage>
<div class="block span4">
    <div class="border">
        <a href="@Url.ContentUrl(Model.CurrentPage.ContentLink)">
            <h2 @Html.EditAttributes(x => x.CurrentPage.Name)>
                @Model.CurrentPage.Name</h2>
            <p class="introduction"
               @Html.EditAttributes(x => x.CurrentPage.MetaDescription)>
               @Model.CurrentPage.MetaDescription</p>
            <div>
                @foreach(string usp in
```

```
                    Model.CurrentPage.UniqueSellingPoints)
                {
                    <small class="label label-info" style="color:white;">@usp</small>
                }
            </div>
            <div @Html.EditAttributes(x => x.CurrentPage.PageImage)>
                <img src="@Url.ContentUrl(Model.CurrentPage.PageImage)" />
            </div>
        </a>
    </div>
</div>
```

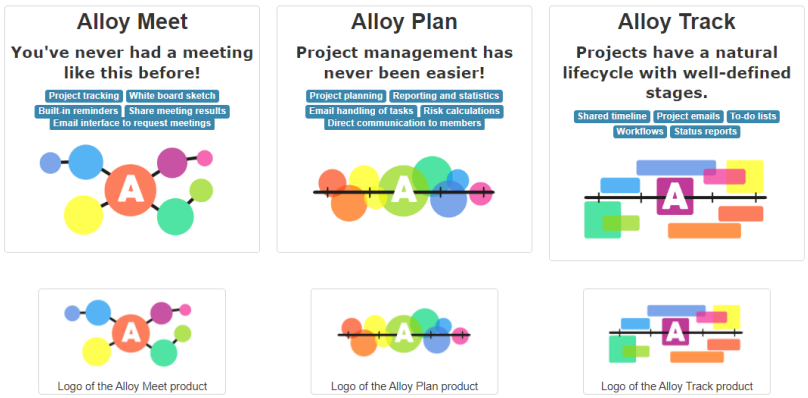
Creating a template view for image files

1. In **AlloyTraining**, in **Views\Shared**, add a new Razor file named **ImageFile.cshtml**.
2. Modify the contents, as shown in the following markup:

```
@using EPiServer.Web.Mvc.Html
@model AlloyTraining.Models.Media.ImageFile

<div class="block span4">
    <figure class="border">
        <img src="@Url.ContentUrl(Model.ContentLink)"
             alt="@Model.Description" />
        <figcaption>@Model.Description</figcaption>
    </figure>
</div>
```

Testing the partial page and image templates

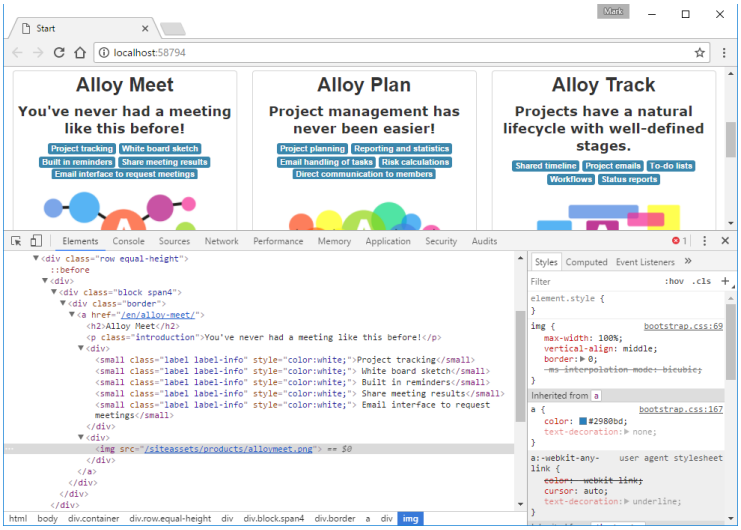
1. Start the **AlloyTraining** website, and note the visitor's view of the content area with three product pages and three product images rendered by their partial content templates:



- If the image descriptions are not showing, then make sure you have published the metadata for them.

2. Click each partial product page to confirm that it links to the correct full product page.
3. Right-click one of the product images, click **Inspect**, and note the following, as shown in the following screenshot:
    - A `<div>` element with Bootstrap class of `row` and `equal-height` that contains three `<div>` elements that were generated for each reference to a product page in the content area.

- Three `<div>` elements with Bootstrap class of `block` and `span4` that were output by the partial view.
- The URL used for the `src` attribute of the product images.



- I am not a CSS or Bootstrap expert, and you might see some strange layouts with the rendered pages and images depending on the current viewport size. Don't worry about this for now.

Creating a partial content template for folders

We might want to be able to add a reference to a folder inside a content area and see its name and how many items are in that folder.

1. In **AlloyTraining**, in **Models/ViewModels**, add a new class named **ContentFolderViewModel.cs**.
2. Modify the statements, as shown in the following code:

```
using EPiServer.Core;

namespace AlloyTraining.Models.ViewModels
{
    public class ContentFolderViewModel
    {
        public ContentFolder CurrentFolder { get; set; }
        public int ItemsCount { get; set; }
    }
}
```

3. In **AlloyTraining**, in **Components**, add a new class named **ContentFolderComponent.cs**.
4. Modify the statements to define a private field to store the `IContentLoader` and set it in the constructor, change the `Index` action method to use a parameter named `currentContent`, and create an instance of the `ContentFolderViewModel` and set its properties, as shown in the following code:

```
using AlloyTraining.Models.ViewModels; // ContentFolderViewModel
using EPiServer; // IContentLoader
using EPiServer.Core; // ContentFolder
using EPiServer.Web.Mvc; // PartialContentComponent<T>
using Microsoft.AspNetCore.Mvc; // IViewComponentResult
```

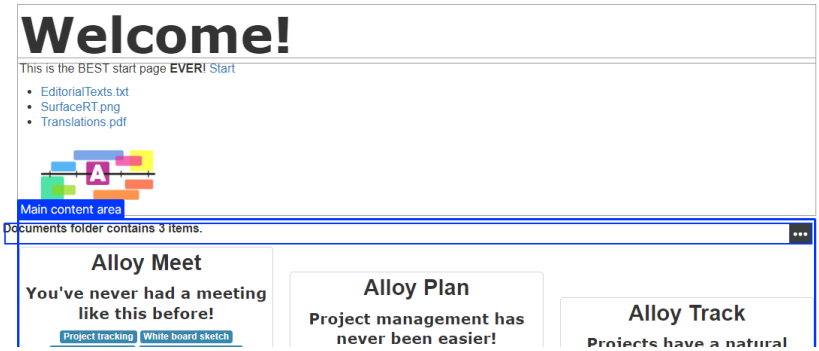
```
using System.Linq; // Count extension method

namespace AlloyTraining.Components
{
    public class ContentFolderComponent : PartialContentComponent<ContentFolder>
    {
        protected readonly IContentLoader loader;

        public ContentFolderComponent(IContentLoader loader)
        {
            this.loader = loader;
        }

        protected override IViewComponentResult InvokeComponent(
            ContentFolder currentContent)
        {
            ContentFolderViewModel viewmodel = new()
            {
                CurrentFolder = currentContent,
                ItemsCount = loader.GetChildren<IContent>
                    (currentContent.ContentLink).Count()
            };

            return View(viewmodel);
        }
    }
}
```

5. In **AlloyTraining**, in **Views\Components**, add a new folder named **ContentFolder**.
6. In **ContentFolder**, add a new Razor view named **Default.cshtml**.
7. Modify the file, as shown in the following markup:

```
@model AlloyTraining.Models.ViewModels.ContentFolderViewModel
<h4>@Model.CurrentFolder.Name folder contains @Model.ItemsCount items.</h4>
```

Testing the partial content template for folders

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Edit the **Start** page, drag and drop the **Documents** folder from the **Assets** pane into the main content area, and note the folder information is displayed, as shown in the following screenshot:



3. Publish the **Start** page.
4. Close the browser and shut down the web server.

Exercise C2 – Creating a partial template for all pages

In this exercise, you will create partial page templates to enable any site page to render inside a content area.

Page templates can be defined for base classes and then inherited by any page type that derives from that base class. You will:

- Define a partial page template for all pages to render the page Name and MetaDescription in a "full" 3/3 width view with yellow background colour.
- Define a partial page template for all pages to render the page Name and MetaDescription in a "wide" 2/3 width view with pink background colour.
- Define a partial page template for all pages to render the page Name and MetaDescription in a "narrow" 1/3 width view with green background colour.
- Define **SiteTags** class with string constants for the three tag values: "Full", "Wide", and "Narrow".

Prerequisites: complete Exercises B1 to B4, C1.

Creating a partial content component for all pages

1. In **AlloyTraining**, add a new class file named **SiteTags.cs**.
2. Modify the file, as shown in the following code:

```
namespace AlloyTraining
{
    public static class SiteTags
    {
        public const string Full = "full";
        public const string Wide = "wide";
        public const string Narrow = "narrow";
    }
}
```

- These string constants will be used to apply tags to content templates.

3. In **AlloyTraining**, in **Components**, copy and paste the **ProductPagePartialComponent.cs** file.
4. Rename the file copy to **AllPagesPartialComponent.cs**.
5. Open **AllPagesPartialComponent.cs**.

- This single file will contain three classes, for **full**, **wide**, and **narrow** templates.

6. Rename the class to `AllPagesFullPartialComponent`.
7. Change all references of `ProductPage` to `SitePageData`.
8. Apply `TemplateDescriptor` attribute to mark this class as allowing page template inheritance.
9. Modify the call to `View` to pass in the name of a view: `SiteTags.Full`

The class should now look something like this:

```
using AlloyTraining.Models.Pages; // SitePageData
using AlloyTraining.Models.ViewModels; // PageViewModel
using EPiServer.Framework.DataAnnotations; // [TemplateDescriptor]
using EPiServer.Web.Mvc; // PartialContentComponent
using Microsoft.AspNetCore.Mvc; // IViewComponentResult

namespace AlloyTraining.Components
{
```

```
    [TemplateDescriptor(Inherited = true,
      Tags = new[] { SiteTags.Full }, AvailableWithoutTag = true)]
    public class AllPagesFullPartialComponent :
       PartialContentComponent<SitePageData>
    {
       protected override IViewComponentResult InvokeComponent(
          SitePageData currentPage)
       {
          return View(viewName: SiteTags.Full,
             model: PageViewModel.Create(currentPage));
       }
    }
}
```

- When you explicitly specify a view name, the search path looks for that named view in **~\Views\Shared\Components\SitePageData\Full.cshtml**, i.e. **~\Views\Shared\Full.cshtml**

Creating additional partial page templates for "wide" and "narrow"

1. Inside the namespace, copy the entire class and its attribute to the clipboard.
2. Paste twice to create two copies of the class.
    a. Rename the first copy to: **AllPagesWidePartialController**
    b. Rename the second copy to: **AllPagesNarrowPartialController**
3. Change the `TemplateDescriptor` attributes of the two copies to set a Tag named "wide" or "narrow" and make the two copies only available if the tag is set.
4. Explicitly pass the name of a partial view to use instead of Full: Wide or Narrow.

- You must set `AvailableWithoutTag = false` for the wide and narrow templates to make sure that they are not used when no tag is set.

Your two copied classes should look something like the following code:

```
    [TemplateDescriptor(Inherited = true,
      Tags = new[] { SiteTags.Wide }, AvailableWithoutTag = false)]
    public class AllPagesWidePartialComponent :
       PartialContentComponent<SitePageData>
    {
       protected override IViewComponentResult InvokeComponent(
          SitePageData currentPage)
       {
          return View(viewName: SiteTags.Wide,
             model: PageViewModel.Create(currentPage));
       }
    }

    [TemplateDescriptor(Inherited = true,
      Tags = new[] { SiteTags.Narrow }, AvailableWithoutTag = false)]
    public class AllPagesNarrowPartialComponent :
       PartialContentComponent<SitePageData>
    {
       protected override IViewComponentResult InvokeComponent(
          SitePageData currentPage)
       {
          return View(viewName: SiteTags.Narrow,
```

```
            model: PageViewModel.Create(currentPage));
    }
}
```

Creating partial views for all pages

1. In **Views\Shared\Components**, create a new folder named **SitePageData**.
2. From the **Views\Shared\Components\ProductPage** folder, copy the **Default.cshtml** file into the folder **Views\Shared\Components\SitePageData** and rename it to **Full.cshtml**.
3. Open **Views\Shared\Components\SitePageData\Full.cshtml**.
4. Modify the contents, as shown in the following markup, and note the following:
   - `PageViewModel<T>` now uses a `SitePageData` instead of `ProductPage`, so that the template will work with any type of page on the site.
   - It has a Bootstrap class of `span12` for "full" width.
   - It has a style that sets the background color to light yellow.
   - It outputs three properties in this order vertically: `Name`, `MetaDescription`, and `PageImage`.

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<SitePageData>
<div class="block span12" style="background-color: lightyellow;">
    <div class="border">
        <a href="@Url.ContentUrl(Model.CurrentPage.ContentLink)">
            <h2 @Html.EditAttributes(x => x.CurrentPage.Name)>
                @Model.CurrentPage.Name</h2>
            <p class="introduction"
                @Html.EditAttributes(x => x.CurrentPage.MetaDescription)>
                @Model.CurrentPage.MetaDescription</p>
            <div @Html.EditAttributes(x => x.CurrentPage.PageImage)>
                <img src="@Url.ContentUrl(Model.CurrentPage.PageImage)" />
            </div>
        </a>
    </div>
</div>
```

5. Save and close **Full.cshtml**.
6. Copy and paste **Full.cshtml** twice:
   - Rename the first copy: **Wide.cshtml**
   - Rename the second copy: **Narrow.cshtml**
7. Open **Wide.cshtml**.
8. Modify the contents, as shown in the following markup, and note the following:
   - It has a Bootstrap class of **span8** for "wide" 2/3 width.
   - It has a style that sets the background color to light pink (lavenderblush).
   - It outputs two properties in this order vertically: **PageImage** and **Name**.

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<SitePageData>
<div class="block span8" style="background-color: lavenderblush;">
    <div class="border">
        <a href="@Url.ContentUrl(Model.CurrentPage.ContentLink)">
            <div @Html.EditAttributes(x => x.CurrentPage.PageImage)>
```

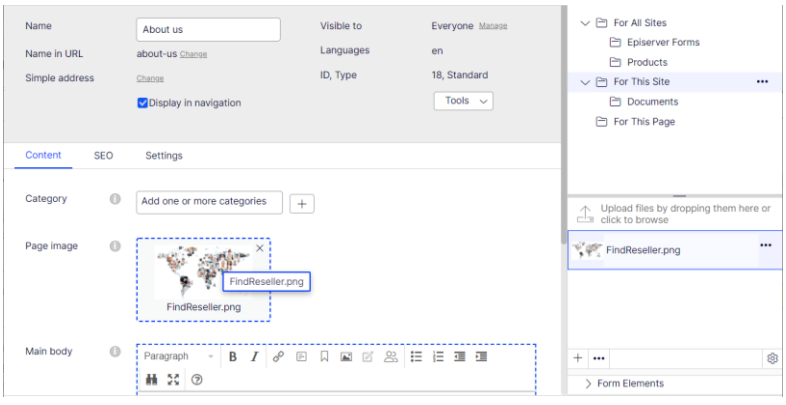
```
                <img src="@Url.ContentUrl(Model.CurrentPage.PageImage)" />
            </div>
            <h2 @Html.EditAttributes(x => x.CurrentPage.Name)>
                @Model.CurrentPage.Name</h2>
        </a>
    </div>
</div>
```

9. Open **Narrow.cshtml**.
10. Modify the contents, as shown in the following markup, and note the following:
    - It has a Bootstrap class of **span4** for "narrow" 1/3 width.
    - It has a style that sets the background color to pale green.
    - It outputs two properties in this order vertically: **Name** and **MetaDescription**.

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<SitePageData>
<div class="block span4" style="background-color: palegreen;">
    <div class="border">
        <a href="@Url.ContentUrl(Model.CurrentPage.ContentLink)">
            <h2 @Html.EditAttributes(x => x.CurrentPage.Name)>
                @Model.CurrentPage.Name</h2>
            <p class="introduction"
                @Html.EditAttributes(x => x.CurrentPage.MetaDescription)>
                @Model.CurrentPage.MetaDescription</p>
        </a>
    </div>
</div>
```

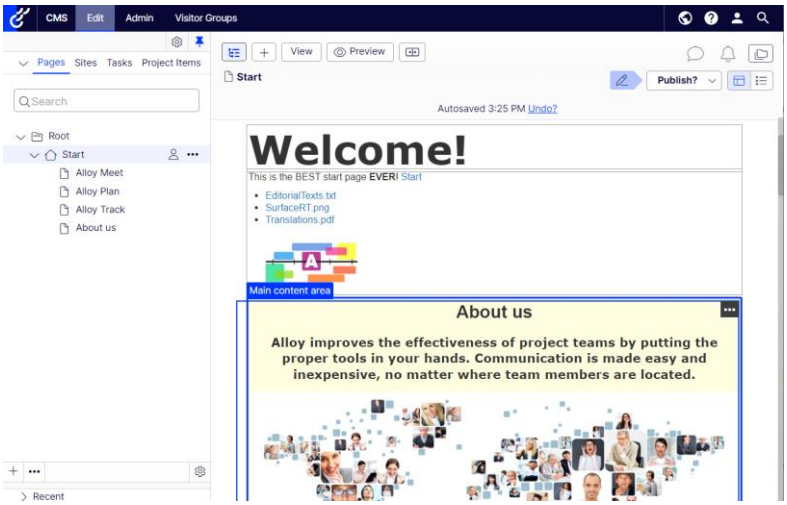
Testing the partial page template

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Edit the **About us** page, and switch to **All Properties** view.
3. In **Assets** pane, click **Media**.
4. Upload the image **~\Assets\Misc\FindReseller.png** to the folder **For This Site**.
5. Drag and drop the image **FindReseller.png** to the **Page image** property of the **About us** page, as shown in the following screenshot:



6. Publish the change.

7. Edit the **Start** page.
8. Drag and drop the **About us** page into the main content area, as shown in the following screenshot, and note that it uses the "full" width partial page template:



9. Close the browser and shut down the web server.

- In the next exercise, you will allow the content editor to choose display options to apply one of three "tags" to switch between the three partial page templates.

Exercise C3 – Enabling editors to apply tags manually using display options

In this exercise, you will define three display options to alloy an editor to apply tags manually to individual content items in a content area to switch between partial page templates.

Prerequisites: complete Exercises B1 to B4, C1 and C2.

Adding display options using an initialization module

Display options for applying tags to content references should be registered during initialization.

1. In **AlloyTraining**, in **Startup.cs**, in the `ConfigureServices` method, add statements to configure a menu of choices for display options, as shown in the following code:

```
services.Configure<DisplayOptions>(options =>
{
    options.Add(id: SiteTags.Full, name: "Full", tag: SiteTags.Full);
    options.Add(id: SiteTags.Wide, name: "Wide", tag: SiteTags.Wide);
    options.Add(id: SiteTags.Narrow, name: "Narrow", tag: SiteTags.Narrow);
});
```

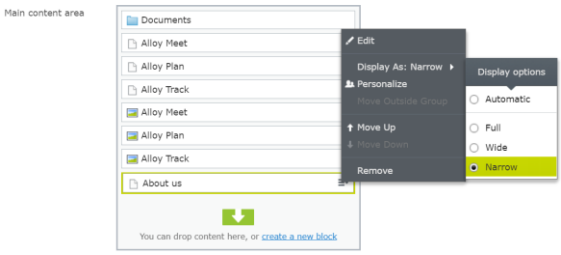
2. In **~\Resources\Translations**, add an XML file named **DisplayOptions.xml**.
3. Modify the file, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<languages>
  <language name="English" id="en">
    <displayoptions>
      <full>Full</full>
      <wide>Wide</wide>
      <narrow>Narrow</narrow>
    </displayoptions>
  </language>
  <language name="Swedish" id="sv">
    <displayoptions>
      <full>Full</full>
      <wide>Bred</wide>
      <narrow>Smal</narrow>
    </displayoptions>
  </language>
</languages>
```

Testing display options

1. Start the **AlloyTraining** website and log in as a CMS admin.
2. Edit the **Start** page and switch to **All Properties** view.
3. Click **Content** tab and scroll down to the **Main content area** property.

4. Select **About us**, and click the context menu to choose a **Display As** option, for example **Narrow**, as shown in the following screenshot:



5. Publish the change and note the partial page template used to render the **About us** page has been changed to use the **Narrow** template (that doesn't show the image and it has a green background).
6. Try applying the **Wide** display option, and note it renders the image above the title, without a description, as shown in the following screenshot:

7. Publish the page.
8. Close the browser and shut down the web server.

- **Display As** options are shown for all content references in a content area, even if they would have no affect. For example, the references to the three images show the same **Display As** options, but the images do not have different templates. The references to the three product pages can have the display options applied because `ProductPage` inherits indirectly from `SitePageData`. It is possible to filter display options based on the content type of the current selection, but that is a technique not covered in this course but you can read about it at the following link: https://getadigital.com/no/blogg/filtered-display-option-menu-based-on-content-type-in-episerver/

Exercise C4 – Applying tags to content areas with code

In this exercise, you will create two content areas on the product page and then control which partial page template is used by programmatically applying a tag: full, wide, or narrow.

Prerequisites: complete Exercises B1 to B4, C1 to C2.

Adding a content area to the Product page type and template

1. In **AlloyTraining**, in **Models\Pages\ProductPage.cs**, add two **ContentArea** properties with appropriate attributes: **MainContentArea** and **RelatedContentArea**, as shown in the following code:

```
[Display(Name = "Main content area",
    Description = "Drag and drop blocks and pages with partial templates.",
    GroupName = SystemTabNames.Content,
    Order = 330)]
public virtual ContentArea MainContentArea { get; set; }

[Display(Name = "Related content area",
    Description = "Drag and drop blocks and pages with partial templates.",
    GroupName = SystemTabNames.Content,
    Order = 340)]
public virtual ContentArea RelatedContentArea { get; set; }
```

2. In **AlloyTraining**, open **Views\ProductPage\Index.cshtml**.
3. Import the **AlloyTraining** namespace, as shown in the following markup:

```
@using AlloyTraining
```

4. Above `@section RelatedContent`, render the **MainContentArea** property so that editors get an on-page editing experience, set Bootstrap classes of **row** and **equal-height**, and set the wide site tag, as shown in the following markup:

```
@Html.PropertyFor(m => m.CurrentPage.MainContentArea,
    new { CssClass = "row equal-height", Tag = SiteTags.Wide })
```

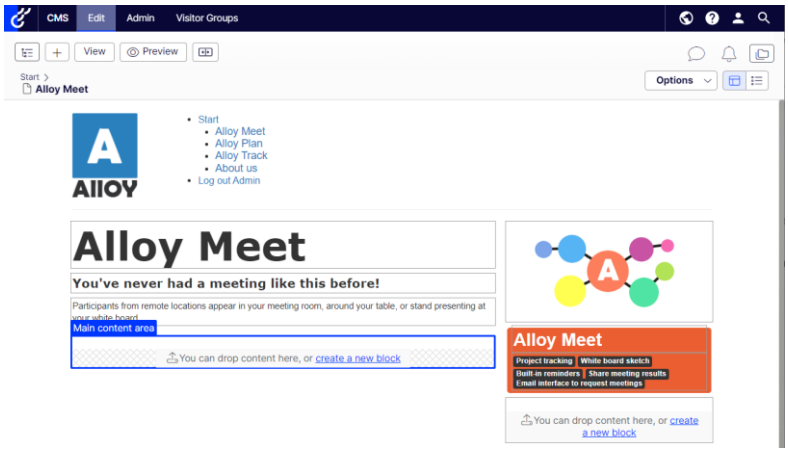
5. At the bottom of `@section RelatedContent`, before the close brace, render the **RelatedContentArea** property so that editors get an on-page editing experience, set Bootstrap classes of **row** and **equal-height**, and set the narrow site tag, as shown in the following markup:

```
@Html.PropertyFor(m => m.CurrentPage.RelatedContentArea,
    new { CssClass = "row equal-height", Tag = SiteTags.Narrow })
```

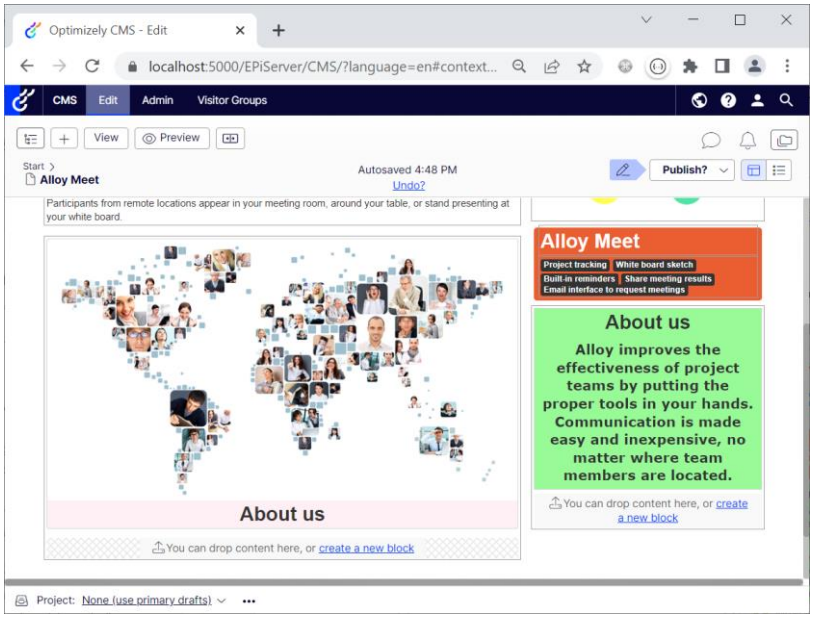
Testing the content areas on product pages

1. Start the **AlloyTraining** website, and log in as a CMS admin.

2. Edit one of the product pages, for example, **Alloy Meet**, as shown in the following screenshot:



3. Drag and drop **About us** from the **Navigation** pane's **Pages** tree, into the **Main content area**.
4. Drag and drop **About us** from the **Navigation** pane's **Pages** tree, into the **Related content area**.
5. Note that different partial page templates are used (**Wide.cshtml** and **Narrow.cshtml**) due to the tags applied programmatically, as shown in the following screenshot:



6. Publish the page.
7. Close the browser and shut down the web server.

Exercise C5 – Applying tags automatically using a display channel

In this exercise, you will create a page template for Start page optimized for mobile devices and a display channel that automatically sets the mobile tag based on information in an incoming HTTP request.

Prerequisites: complete Exercises B1 – B4.

Creating a mobile page template controller and view

1. In **AlloyTraining**, in **Controllers**, copy and paste the **StartPageController.cs** file.
2. Rename the copy to **StartPageMobileController.cs**, and open it.
3. Import the **EPiServer.Framework.Web** namespace, as shown in the following code:

```
using EPiServer.Framework.Web;
```

4. Apply parameters to the `TemplateDescriptor` attribute to ensure this controller is only used when the "mobile" rendering tag is set, as shown in the following code:

```
[TemplateDescriptor(Inherited = true, AvailableWithoutTag = false,
    Tags = new[] { EPiServer.Framework.Web.RenderingTags.Mobile })]
public class StartPageMobileController : PageControllerBase<StartPage>
```

5. In **AlloyTraining**, in **Views**, add a new folder named **StartPageMobile**.
6. In **Views\StartPage**, copy the **Index.cshtml** file into the folder **Views\StartPageMobile**.
7. In **Views\StartPageMobile\Index.cshtml**, at the bottom of the view, replace the outputting of the **MainContentArea** using **PropertyFor** with an enumeration of the filtered items in the content area that outputs a link to each page, as shown in the following markup:

```
<ul>
    @foreach(var item in Model.CurrentPage.MainContentArea.FilteredItems)
    {
        <li>@Html.ContentLink(item.ContentLink)</li>
    }
</ul>
```

- **FilteredItems** removes any content references that the current visitor should not be able to see.

Creating a display channel to automatically select the mobile template

ASP.NET had an easy built-in object for detecting mobile browsers. You could just use `HttpContext.Browser.IsMobileDevice`. This is not available in ASP.NET Core. We will use an open source library to provide similar detection: https://www.nuget.org/packages/wangkanai.detection

1. In **AlloyTraining**, add a package reference to `Wangkanai.Detection`, as shown in the following markup:

```
<PackageReference Include="Wangkanai.Detection" Version="5.2.0" />
```

2. Build the **AlloyTraining** project to restore NuGet packages.
3. In **AlloyTraining**, in **Business**, add a new folder named **DisplayChannels**.
4. In **Business\DisplayChannels**, add a new class file named **MobileDisplayChannel.cs**.
5. Import the **EPiServer.Web** namespace and make the class inherit from **DisplayChannel**.
6. Use Visual Studio to implement the abstract class.
7. For the **ChannelName** property, return the mobile rendering tag.
8. For the **IsActive** method, return if the current request is from a mobile device.

Your completed class should look something like the following code:

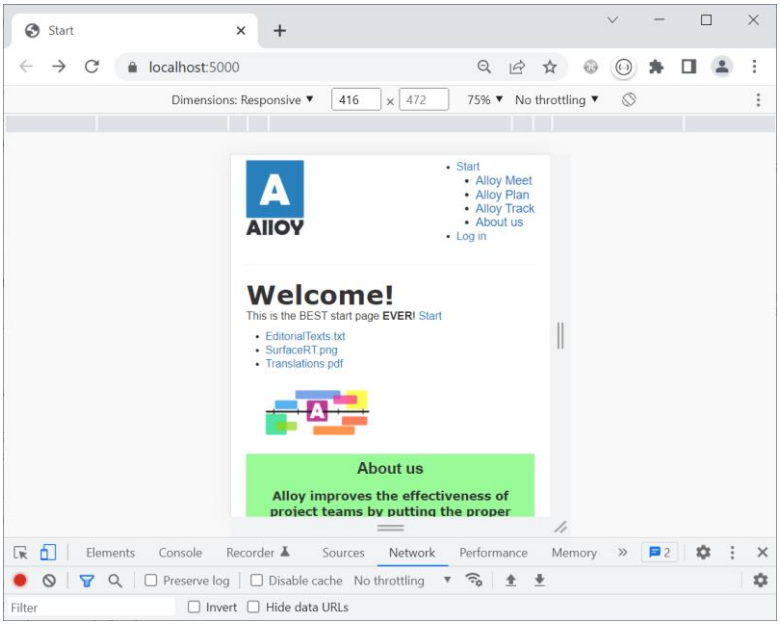
```
using EPiServer.Framework.Web; // RenderingTags
using EPiServer.Web; // DisplayChannel
using Microsoft.AspNetCore.Http; // HttpContext
using Microsoft.Extensions.DependencyInjection;
using Wangkanai.Detection.Models; // Device
using Wangkanai.Detection.Services; // IDetectionService

namespace AlloyTraining.Business.DisplayChannels
{
    public class MobileDisplayChannel : DisplayChannel
    {
        public override string ChannelName => RenderingTags.Mobile;

        public override bool IsActive(HttpContext context)
        {
            var detection = context.RequestServices
                .GetRequiredService<IDetectionService>();
            return detection.Device.Type == Device.Mobile;
        }
    }
}
```

8. In **Startup.cs**, in `ConfigureServices`, add a call to add detection, as shown in the following code:

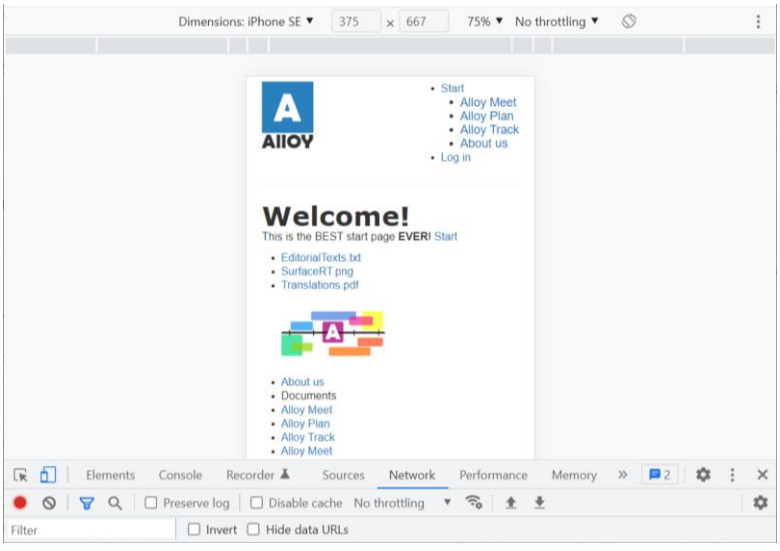
```
services.AddDetection();
```

Testing the display channel

1. Start the **AlloyTraining** website.
2. Press *F12* to show developer tools.
3. In Chrome's developer tools pane, click **Toogle device toolbar**, or press *Ctrl* + *Shift* + *M*, and note that by default Chrome shows a **Responsive** page, as shown in the following screenshot:

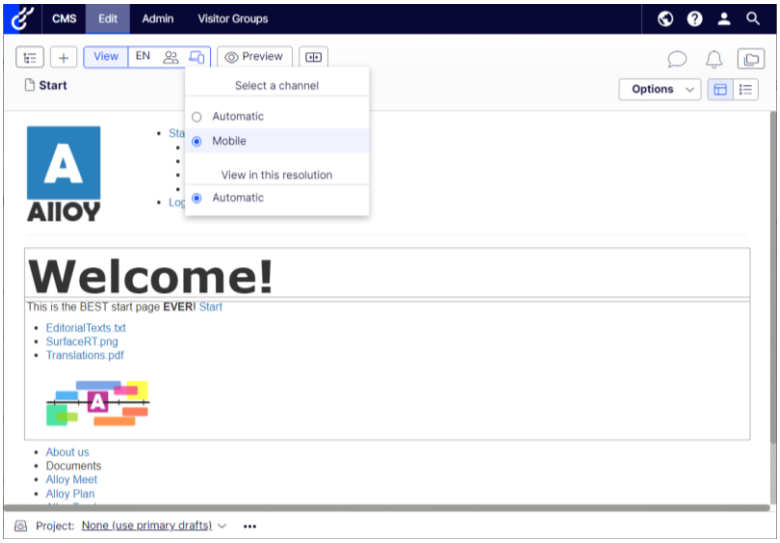


4. In the device toolbar, choose **iPhone SE**, click **Refresh** or press *F5*, and note Chrome now shows the "mobile" page template response with links instead of partial content, as shown in the following screenshot:



5. Close the developer tools.
6. Log in as a CMS admin.
7. Edit the **Start** page.

8. In the toolbar, click **Toggle view settings**, and select the **mobile** channel, as shown in the following screenshot:



9. Close the browser and shut down the web server.

Exercise C6 – Disabling On-Page Edit view

In this exercise, you will use an interface to disable On-Page Edit view for pages that implement it.

Prerequisites: complete Exercises B1 to B4.

Defining an interface and class for disabling On-Page Edit view

1. In **AlloyTraining**, in the **Business** folder, create a folder named **EditorDescriptors**.
2. In the **Business\EditorDescriptors** folder, add a new interface named **IDisableOnPageEditView.cs**.
3. The interface does not need any members, so remove any `using` statements, and make it `public`, as shown in the following code:

```
namespace AlloyTraining.Business.EditorDescriptors
{
    public interface IDisableOnPageEditView
    {
    }
}
```

4. In the **Business\EditorDescriptors** folder, add a new class named **DisableOnPageEditViewEditorDescriptor.cs**.
5. Modify the statements to define a class that inherits from `UIDescriptor<IDisableOnPageEditView>` and disables On-Page Edit view (and a couple of others), as shown in the following code:

```
// [UIDescriptorRegistration], UIDescriptor, CmsViewNames
using EPiServer.Shell;
using System.Collections.Generic; // List<T>

namespace AlloyTraining.Business.EditorDescriptors
{
    [UIDescriptorRegistration]
    public class DisableOnPageEditViewEditorDescriptor
      : UIDescriptor<IDisableOnPageEditView>
    {
        public DisableOnPageEditViewEditorDescriptor()
        {
            DisabledViews = new List<string>
            {
                CmsViewNames.OnPageEditView,
                CmsViewNames.PreviewView,
                CmsViewNames.ContentListingView
            };

            DefaultView = CmsViewNames.AllPropertiesView;
        }
    }
}
```

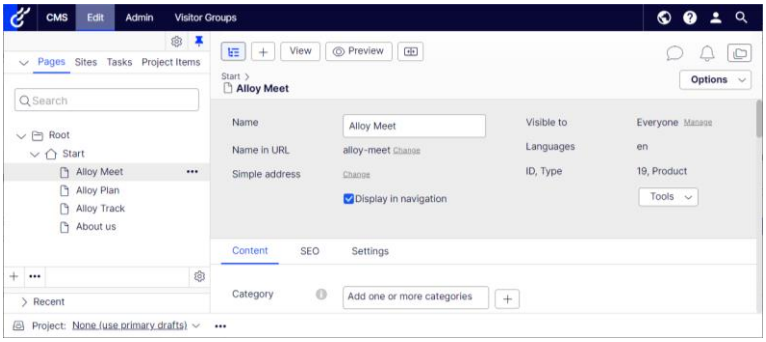
6. In **Models/Pages/ProductPage.cs**, modify the statements to import the `AlloyTraining.Business.EditorDescriptors` namespace and then implement the `IDisableOnPageEditView` interface, as shown in the following code:

```
public class ProductPage : StandardPage, IDisableOnPageEditView
```

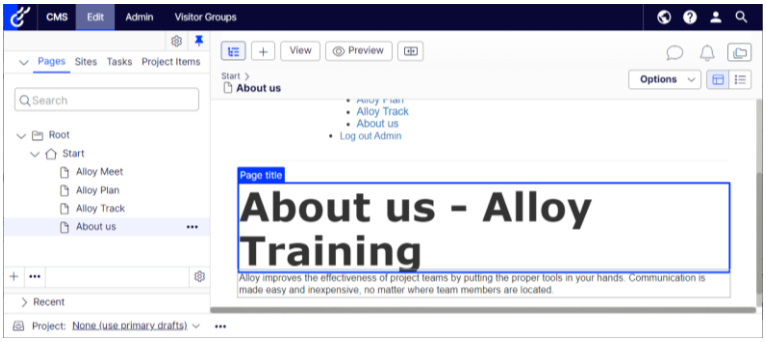
Testing disabling On-Page Edit view

1. Start the **AlloyTraining** website, and log in as a CMS admin.

2. On the **Start** page, switch to **Edit** view, and note the two buttons to toggle between **On-Page Edit** view and **All Properties** view.
3. In the **Pages** tree, select any product page and note the two buttons are gone and **All Properties** view is on by default, as shown in the following screenshot:



4. In the **Pages** tree, select the **About us** page and note the two buttons to toggle between **On-Page Edit** view and **All Properties** view, as shown in the following screenshot:



5. Close the browser and shut down the web server.
6. Open **Models/Pages/ProductPage.cs**.
7. Comment out the implementation of the interface to return product page editing behavior back to normal.

Module D – Working with Blocks

Goal

The overall goal of the exercises in this module is to implement working examples of various block types and uses for blocks. You will:

1. Create a controller-less block for efficiency.
2. Create a block with a controller.
3. Create a preview template for blocks and partial pages.
4. Move important block properties to the basic info area.
5. Use a block type as a property type.

Exercise D1 – Creating a controller-less block for editorial content

In this exercise, you will create a block type with a block template that consists of a view without a controller.

Controller-less blocks are more efficient than block templates with controllers, so you should use them whenever possible.

Prerequisites: complete Exercises B1 to B4, C1 to C4.

- If you have not completed Exercises C1 to C4, then you can just complete the task, **Adding a content area to the Product page type and template** in Exercise C4 on page 150, to define two content areas on a product page, and then you can complete exercise D1.

Creating an editorial block type

We will define a simple block type with template that does not need a controller for maximum performance:

1. In **AlloyTraining**, in **Models**, add a new folder named **Blocks**.
2. In **Blocks**, add a new class file named **EditorialBlock.cs**.
3. Decorate the `EditorialBlock` class with the `[ContentType]` attribute, change the **DisplayName** to **Editorial**, add a **Description** of "Use this for a rich editorial text that will be reused in multiple places.", apply the **[SiteBlockIcon]** attribute to the class, and add an **XhtmlString** property named **MainBody**, and allow it to be localized into multiple languages, so the class looks something like the following code:

```
using EPiServer.Core; // BlockData, XhtmlString
using EPiServer.DataAbstraction; // SystemTabNames
using EPiServer.DataAnnotations; // [ContentType], [CultureSpecific]
using System.ComponentModel.DataAnnotations; // [Display]

namespace AlloyTraining.Models.Blocks
{
    [ContentType(DisplayName = "Editorial",
        GroupName = SiteGroupNames.Common,
        Description = "Use this for a rich editorial text that will be reused in multiple places.")]
    [SiteBlockIcon]
    public class EditorialBlock : BlockData
    {
        [CultureSpecific]
        [Display(Name = "Main body",
```

```
        Description = "The main body will be shown in the main content area of the page, using the
XHTML-editor you can insert for example text, images and tables.",
        GroupName = SystemTabNames.Content,
        Order = 10)]
    public virtual XhtmlString MainBody { get; set; }
  }
}
```

Creating a controller-less block template

We will follow location and naming conventions to allow the block template to be found automatically:

1. In **AlloyTraining**, in **Views\Shared**, add an empty Razor view named **EditorialBlock.cshtml**.
2. Modify the view to use `EditorialBlock` as its model and to output the `MainBody` property to support on-page editing, as shown in the following markup:

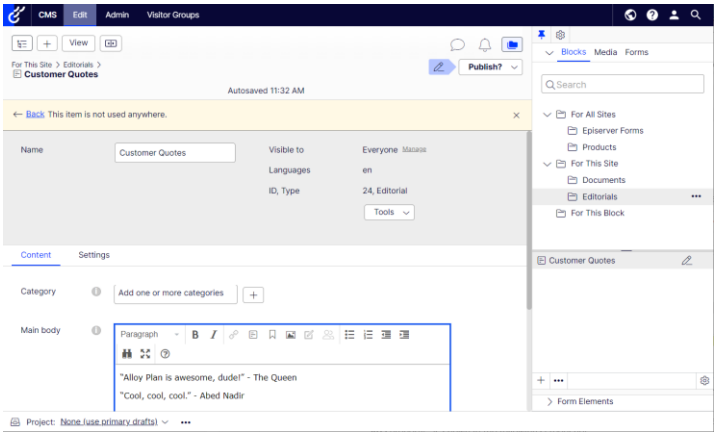
```
@model AlloyTraining.Models.Blocks.EditorialBlock
<div class="block span">
    @Html.PropertyFor(m => m.MainBody)
</div>
```

Creating an instance of the block and using it in multiple places

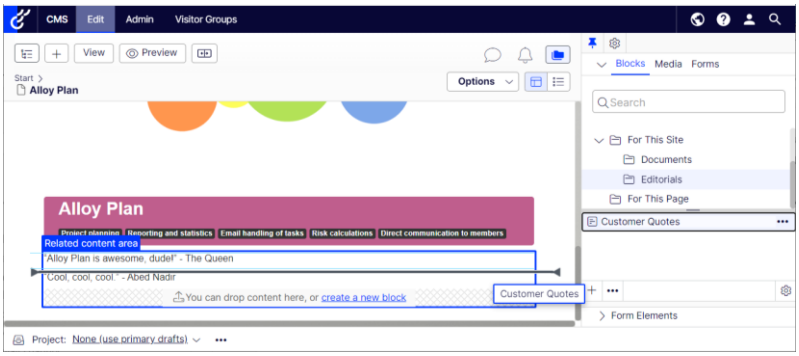
Now we can test the block type and template:

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. In **Assets** pane, click **Blocks**.
3. Add a new folder named **Editorials** to the **For This Site** folder.
4. In the **Editorials** folder, add a new **Editorial** block named **Customer Quotes**, as shown in the following screenshot:

5. In the **Main body** write some quotes from Alloy's customers, like "Alloy Plan is awesome, dude!" - The Queen and "Cool, cool, cool." - Abed Nadir, as shown in the following screenshot:

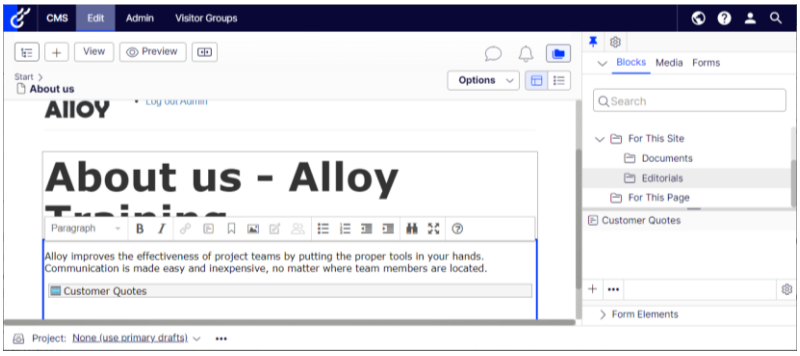


6. Publish the block and note that it is not current used anywhere.
7. Edit the **Alloy Plan** page, and drag and drop the **Customer Quotes** block into its **Related content area** property, as shown in the following screenshot:

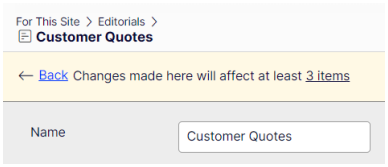


8. Publish the page.
9. Edit the **Alloy Track** page and drag and drop the **Customer Quotes** block into its **Related content area** property.
10. **Publish** the page.

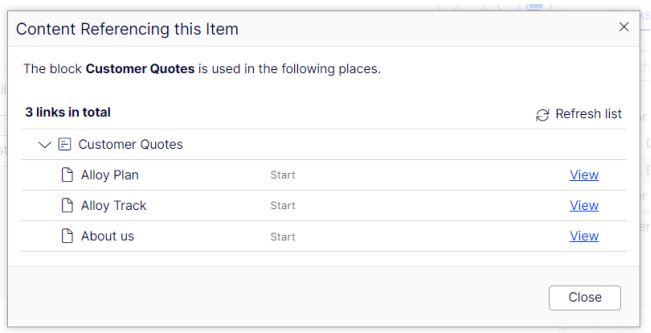
11. Edit the **About us** page, and drag and drop the **Customer Quotes** block into its **Main body** property, as shown in the following screenshot:



12. **Publish** the page.
13. In the **Assets** pane, double-click **Customer Quotes**, and note the block now shows that it is referenced on three items, as shown in the following screenshot:



14. Click **3 items**, and note it shows the pages that reference the shared block, as shown in the following screenshot:



15. Close the browser and shut down the web server.

Exercise D2 – Creating a block with a controller for teaser content

In this exercise, you will create a block type with a block template that is a view with a controller.

Blocks with controllers are more resource hungry, so you should only use them when necessary. For example, when content stored in the CMS must be combined with data from a web service, or a calculated value. In this example, we will generate a random number for a visitor count that will be shown in the block.

Prerequisites: complete Exercises B1 to B4, C1 to C4.

- If you have not completed Exercises C1 to C4, then you can just complete the task, **Adding a content area to the Start page type and template** in Exercise C1 on page 137, to define two content areas on a product page, and then you can complete exercise D2.

Creating a teaser block type

1. In **AlloyTraining**, in **Models**, in **Blocks**, add a new class file named **TeaserBlock.cs**.
2. In **TeaserBlock.cs**, apply the `[ContentType]` attribute to the class, change the **DisplayName** to **Teaser**, add a **Description** of "Use this for rich text with heading, image and page link that will be reused in multiple places.", apply the `[SiteBlockIcon]` attribute to the class, and add the following four properties with appropriate attributes:
    - **TeaserHeading**: `string` (with support for multiple languages)
    - **TeaserText**: `XhtmlString` (with support for multiple languages)
    - **TeaserImage**: `ContentReference` (images only)
    - **TeaserLink**: `PageReference`

Your code should look something like the following:

```
using EPiServer.Core; // BlockData, XhtmlString, ContentReference, PageReference
using EPiServer.DataAnnotations; // [ContentType], [CultureSpecific]
using EPiServer.Web; // UIHint
using System.ComponentModel.DataAnnotations; // [Display], [UIHint]

namespace AlloyTraining.Models.Blocks
{
    [ContentType(DisplayName = "Teaser",
        GroupName = SiteGroupNames.Common,
        Description = " Use this for rich text with heading, image and page link that will be reused in multiple
places.")]
    [SiteBlockIcon]
    public class TeaserBlock : BlockData
    {
        [CultureSpecific]
        [Display(Name = "Heading", Order = 10)]
        public virtual string TeaserHeading { get; set; }

        [CultureSpecific]
        [Display(Name = "Rich text", Order = 20)]
        public virtual XhtmlString TeaserText { get; set; }

        [Display(Name = "Image", Order = 30)]
        [UIHint(UIHint.Image)]
        public virtual ContentReference TeaserImage { get; set; }

        [Display(Name = "Link", Order = 40)]
```

```
    public virtual PageReference TeaserLink { get; set; }
  }
}
```

Creating a teaser view model, controller, and view

For a complex content type, its content template is often a combination of a controller, a view model, and a view.

1. In **AlloyTraining**, in **ViewModels**, add a new class file named **TeaserBlockViewModel.cs**.
2. Modify the view model to have two properties: **CurrentBlock** and **VisitorCount**, as shown in the following code:

```
using AlloyTraining.Models.Blocks;

namespace AlloyTraining.Models.ViewModels
{
    public class TeaserBlockViewModel
    {
        public TeaserBlock CurrentBlock { get; set; }
        public int TodaysVisitorCount { get; set; }
    }
}
```

3. In **AlloyTraining**, in **Components**, add a new class file named **TeaserBlockComponent.cs**.
4. Modify the class to create an instance of the teaser view model and set its properties, before passing it to a partial view, as shown in the following code:

```
using AlloyTraining.Models.Blocks; // TeaserBlock
using AlloyTraining.Models.ViewModels; // TeaserBlockViewModel
using EPiServer.Web.Mvc; // BlockComponent<T>
using Microsoft.AspNetCore.Mvc; // IViewComponentResult
using System; // Random

namespace AlloyTraining.Components
{
  public class TeaserBlockComponent : BlockComponent<TeaserBlock>
  {
    protected override IViewComponentResult InvokeComponent(
      TeaserBlock currentBlock)
    {
      TeaserBlockViewModel viewmodel = new()
      {
        CurrentBlock = currentBlock,
        TodaysVisitorCount = new Random().Next(300, 900)
      };
      return View(viewmodel);
    }
  }
}
```

- We have simulated some data for the visitor count using the `Random` class.

5. In **AlloyTraining**, in **Views\Shared\Components**, add a new folder named **TeaserBlock**.
6. In the **TeaserBlock** folder, add a new empty Razor view named **Default.cshtml**.
7. Modify the contents of the view, as shown in the following markup:

```
@model AlloyTraining.Models.ViewModels.TeaserBlockViewModel
```

```
<div class="media">
    <div style="clear:both;">
        <div class="mediaImg">
            @Html.PropertyFor(m => m.CurrentBlock.TeaserImage)
        </div>
        <div class="mediaText">
            <h2 @Html.EditAttributes(m => m.CurrentBlock.TeaserHeading)>
                @Model.CurrentBlock.TeaserHeading
            </h2>
            <p>@Html.PropertyFor(m => m.CurrentBlock.TeaserText)</p>
            <small>There have been @Model.TodaysVisitorCount visitors today.</small>
        </div>
    </div>
</div>
```

Creating an instance of the teaser block

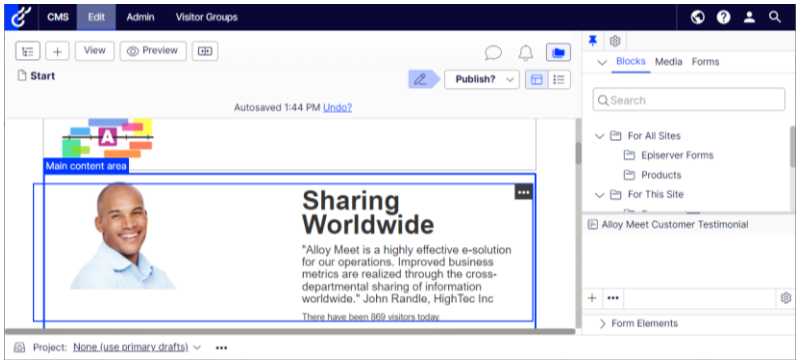
Now, we can test the teaser block:

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. In **Assets** pane, click **Media**.
3. In the **For This Site** folder, add a new folder named **People**.
4. Upload the three images of people in the **Assets\People** folder.
5. In **Assets** pane, click **Blocks**.
6. In the **For This Site** folder, add a new folder named **Teasers**.
7. In the **Teasers** folder, add a new **Teaser** block named **Alloy Meet Customer Testimonial**.
8. Add values for the properties:
    a. **Heading**: Sharing Worldwide
    b. **Rich text**: "Alloy Meet is a highly effective e-solution for our operations. Improved business metrics are realized through the cross-departmental sharing of information worldwide." John Randle, HighTec Inc
    c. **Image**: JohnRandle.jpg (use the picker or drag and drop from Assets pane)
    d. **Link**: Alloy Meet (use the picker or drag and drop from Navigation pane)

- By default, blocks do not support on-page editing, so content editors must use **All Properties** view. In the next exercise, you will create a preview template for blocks to enable an on-page editing experience.

9. **Publish** the block.

10. Edit the **Start** page, drag and drop the **Alloy Meet Customer Testimonial** to the top of its main content area, and publish the page, as shown in the following screenshot:



- This implementation of the teaser block is okay, but it has two limitations: (1) the context menu shows the display options that you defined for product pages, but the teaser block ignores the selection a content editor might make, and (2) editors must use **All Properties** view to change its properties. In the next exercise, you will fix both limitiations.

Exercise D3 – Creating a preview renderer for partial pages and shared blocks

In this exercise, you will create preview renderer for the teaser block.

The renderer will give previews of how the block will look when rendered with various templates.

Prerequisites: complete Exercises B1 – B4, C1 – C4, D2.

Creating three teaser block templates for full, wide, and narrow

1. In **AlloyTraining**, in **Components**, open **TeaserBlockComponent.cs**.
2. Import the EPiServer.Framework.DataAnnotations namespace.
3. Copy and paste the statements that define the **TeaserBlockComponent** class so that you have three controllers, and apply **TemplateDescriptor** to make the template resolver select them when the tags: “full”, “wide”, and “narrow” have been applied, as shown in the following code:

```
using AlloyTraining.Models.Blocks; // TeaserBlock
using AlloyTraining.Models.ViewModels; // TeaserBlockViewModel
using EPiServer.Framework.DataAnnotations; // [TemplateDescriptor]
using EPiServer.Web.Mvc; // BlockComponent<T>
using Microsoft.AspNetCore.Mvc; // IViewComponentResult
using System; // Random

namespace AlloyTraining.Components
{
    [TemplateDescriptor(Tags = new[] { SiteTags.Full },
        AvailableWithoutTag = true)]
    public class TeaserBlockComponent : BlockComponent<TeaserBlock>
    {
        protected override IViewComponentResult InvokeComponent(
            TeaserBlock currentBlock)
        {
            TeaserBlockViewModel viewmodel = new()
            {
                CurrentBlock = currentBlock,
                TodaysVisitorCount = new Random().Next(300, 900)
            };
            return View(viewmodel);
        }
    }

    [TemplateDescriptor(Tags = new[] { SiteTags.Wide })]
    public class TeaserBlockWideComponent : BlockComponent<TeaserBlock>
    {
        protected override IViewComponentResult InvokeComponent(
            TeaserBlock currentBlock)
        {
            TeaserBlockViewModel viewmodel = new()
            {
                CurrentBlock = currentBlock,
                TodaysVisitorCount = new Random().Next(300, 900)
            };
            return View(viewName: "wide", model: viewmodel);
        }
```

```
    }

    [TemplateDescriptor(Tags = new[] { SiteTags.Narrow })]
    public class TeaserBlockNarrowComponent : BlockComponent<TeaserBlock>
    {
        protected override IViewComponentResult InvokeComponent(
            TeaserBlock currentBlock)
        {
            TeaserBlockViewModel viewmodel = new()
            {
                CurrentBlock = currentBlock,
                TodaysVisitorCount = new Random().Next(300, 900)
            };
            return View(viewName: "narrow", model: viewmodel);
        }
    }
}
```

4. In **Views\Shared\Components\TeaserBlock** folder, copy and paste the **Default.cshtml** file twice to create two copies and then rename them to **Wide.cshtml** and **Narrow.cshtml**.
5. Open **Views\Shared\Components\TeaserBlock\Narrow.cshtml**.
6. Add class **span4** to the outer `<div>` and delete the `<div>` that outputs the image.
7. Open **Views\Shared\Components\TeaserBlock\Wide.cshtml**.
8. Add class **span8** to the outer `<div>`.

Creating a preview view model, controller, and view

When a content editor edits the teaser block, we would like them to see a preview of what the block would look like if any of the three display options are selected. To do this, we need to simulate a page that has a property that contains an instance of a block. We can do this by defining a view model with a `ContentArea` and add the instance to the area. Then on the simulated page, we can easily output the block by calling `PropertyFor` three times, once for each display option.

1. In **AlloyTraining**, in **Models\ViewModels**, add a new class file named **PreviewPageViewModel.cs**.
2. Modify the contents, as shown in the following code:

```
using AlloyTraining.Models.Pages; // SitePageData
using EPiServer.Core; // IContent, ContentArea

namespace AlloyTraining.Models.ViewModels
{
    public class PreviewPageViewModel : PageViewModel<SitePageData>
    {
        public PreviewPageViewModel(SitePageData currentPage,
            IContent contentToPreview) : base(currentPage)
        {
            PreviewContentArea.Items.Add(new ContentAreaItem
                { ContentLink = contentToPreview.ContentLink });
        }

        public ContentArea PreviewContentArea { get; set; }
          = new ContentArea();
    }
}
```

3. In **AlloyTraining**, in **Controllers**, add a new class file named **PreviewPageController.cs**.
4. Modify its contents, as shown in the following code:

```
using AlloyTraining.Models.Pages; // SitePageData
using AlloyTraining.Models.ViewModels; // PreviewPageViewModel
using EPiServer; // IContentLoader
using EPiServer.Core; // BlockData, IContent, ContentReference
using EPiServer.Framework.DataAnnotations; // [TemplateDescriptor]
using EPiServer.Framework.Web; // TemplateTypeCategories, RenderingTags
using EPiServer.Web; // IRenderTemplate<T>
using EPiServer.Web.Mvc; // ActionControllerBase
using Microsoft.AspNetCore.Mvc; // IActionResult

namespace AlloyTraining.Controllers
{
    [TemplateDescriptor(Inherited = true,
        TemplateTypeCategory = TemplateTypeCategories.MvcController,
        Tags = new[] { RenderingTags.Preview, RenderingTags.Edit },
        AvailableWithoutTag = false)]
    public class PreviewPageController :
        ActionControllerBase, IRenderTemplate<BlockData>
    {
        protected readonly IContentLoader loader;

        public PreviewPageController(IContentLoader loader)
        {
            this.loader = loader;
        }

        public IActionResult Index(IContent currentContent)
        {
            SitePageData startPage = loader.Get<SitePageData>
                (ContentReference.StartPage);
            PreviewPageViewModel viewmodel = new(
                startPage, currentContent);
            return View(viewmodel);
        }
    }
}
```

- Our site enforces a rule for view models passed to layouts: they must have a `CurrentPage` property with a page instance in it. For our preview, we can get the Start page and pass that in. The preview won't use it, but some page must be passed in.

5. In **AlloyTraining**, in **Views**, add a new folder named **PreviewPage**.
6. In the **PreviewPage** folder, add a new empty Razor view named **Index.cshtml**.
7. Modify the view, as shown in the following markup:

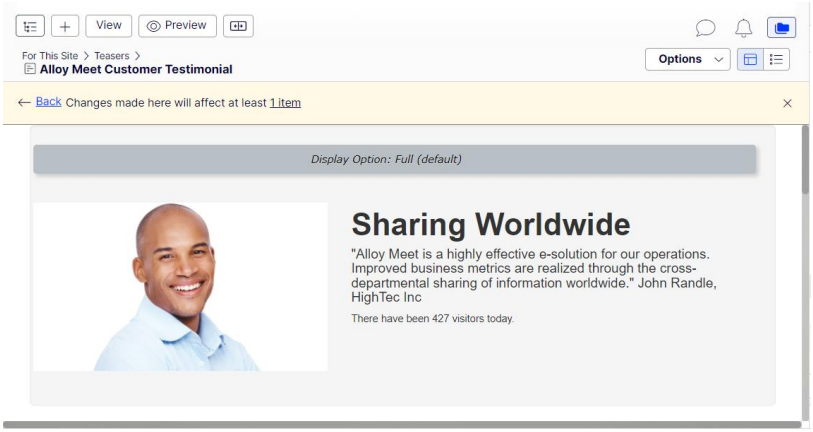
```
@using AlloyTraining
@using AlloyTraining.Models.ViewModels
@using EPiServer.Web.Mvc.Html
@model PreviewPageViewModel
@{
    Layout = null;
}
<head>
```

```
    <title>@(Model.CurrentPage.MetaTitle ?? Model.CurrentPage.Name)</title>
    <link rel="stylesheet" href="@Url.Content("~/css/bootstrap.css")" />
    <link rel="stylesheet"
          href="@Url.Content("~/css/bootstrap-responsive.css")" />
    <link rel="stylesheet" href="@Url.Content("~/css/media.css")" />
    <link rel="stylesheet" href="@Url.Content("~/css/style.css")" />
    <link rel="stylesheet" href="@Url.Content("~/css/editmode.css")" />
    <script type="text/javascript"
            src="@Url.Content("~/js/jquery.js")"></script>
    <script type="text/javascript"
            src="@Url.Content("~/js/bootstrap.js")"></script>
</head>
<body>
    <div class="container">
        <div class="well well-large">
            <div class="row">
                <div class="alert alert-info">Display Option: Full (default)</div>
            </div>
            <div class="row">
                @Html.PropertyFor(m => m.PreviewContentArea,
                  new { Tag = SiteTags.Full })
            </div>
        </div>
        <div class="well well-large">
            <div class="row">
                <div class="alert alert-info span8">Display Option: Wide</div>
            </div>
            <div class="row">
                @Html.PropertyFor(m => m.PreviewContentArea,
                  new { Tag = SiteTags.Wide })
            </div>
        </div>
        <div class="well well-large">
            <div class="row">
                <div class="alert alert-info span4">Display Option: Narrow</div>
            </div>
            <div class="row">
                @Html.PropertyFor(m => m.PreviewContentArea,
                  new { Tag = SiteTags.Narrow })
            </div>
        </div>
    </div>
</body>
```

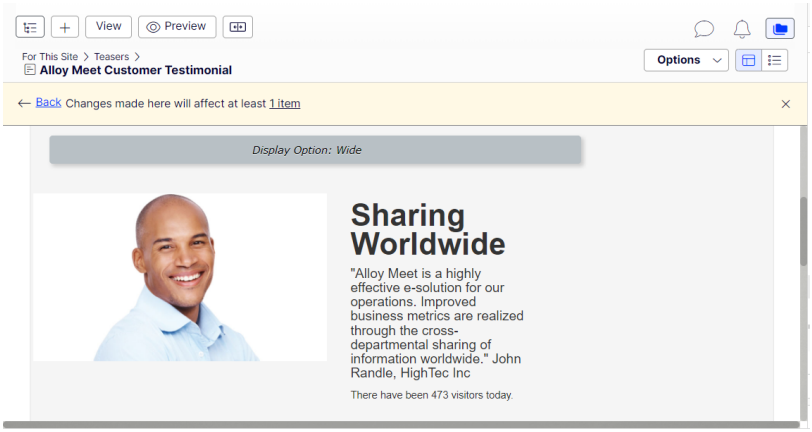
Testing the preview

Now we can test the preview page:

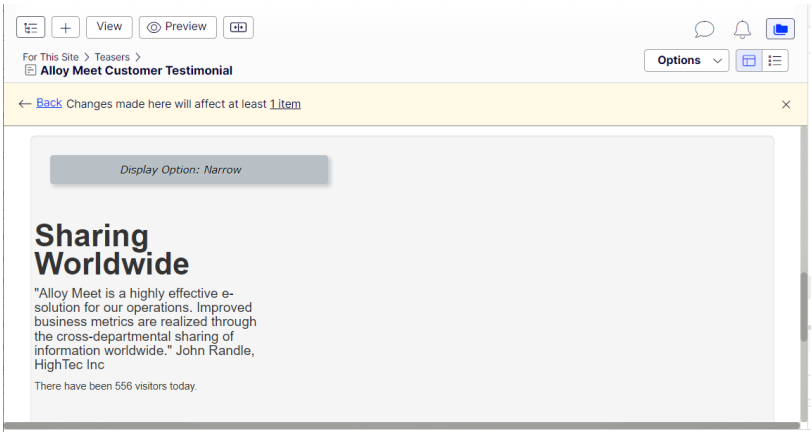
1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Edit the **Alloy Meet Customer Testimonial** teaser block, and note that instead of only allowing **All Properties** view, it defaults to using an **On-Page Editing** view, and shows a preview of the **Full** (default) display option, as shown in the following screenshot:



3. Scroll down the preview page to see the **Wide** display option, as shown in the following screenshot:



4. Scroll down the preview page to see the **Narrow** display option, as shown in the following screenshot:



5. Close the browser and shut down the web server.

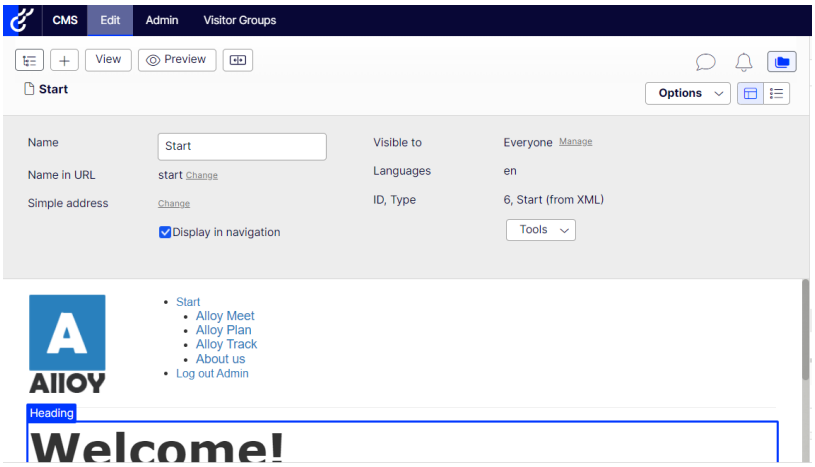
Exercise D4 – Moving properties to the basic info area

In this exercise, you will move a property from the tabbed area up to the basic info properties area.

Prerequisites: complete Exercises B1 – B4, C1 – C4, D2.

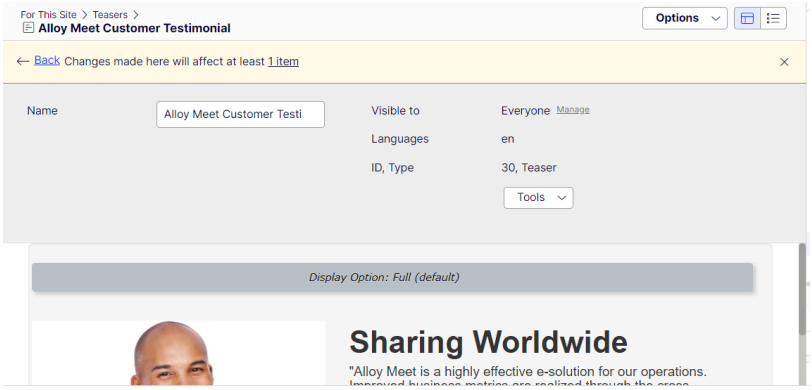
Understanding the existing basic info area

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Edit the **Start** page. Note the on-page editing view contains several properties which are reached by scrolling beyond the top of the page to reveal the top gray area, as shown in the following screenshot. These are called basic info properties and can be used to:
    - Give the page a simple address.
    - Set access rights for a page.
    - Change the name in the URL, and so on.



- This basic info area is always displayed in the **All Properties** editing view.

3. Edit the **Alloy Meet Customer Testimonial** teaser and view the basic information area. Note for a block, like **TeaserBlock**, it cannot have a URL, a simple address, or display in navigation, so there is empty space available that we can make better use of, as shown in the following screenshot:



9. Close the browser and shut down the web server.

Making use of space in the basic information area for blocks

1. In **AlloyTraining**, in **Models/Blocks/TeaserBlock.cs**, import the namespace for the `SystemTabNames` class, as shown in the following code:

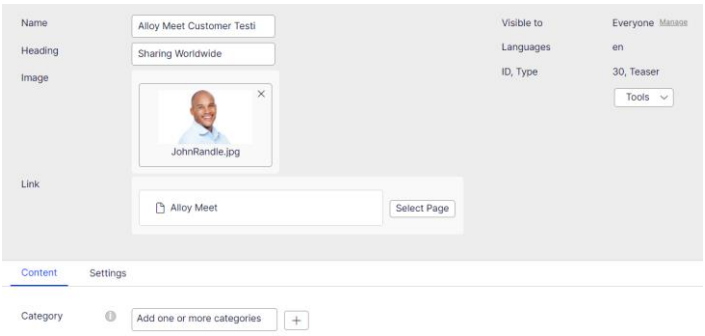
```
using EPiServer.DataAbstraction; // SystemTabNames
```

2. Modify the **[Display]** attribute for the **TeaserHeading**, **TeaserImage**, and **TeaserLink** properties to set the **GroupName** to **PageHeader**, as shown in the following code:

```
[Display(Name = "Link", Order = 10,
    GroupName = SystemTabNames.PageHeader)]
public virtual PageReference TeaserHeading { get; set; }
```

- The "magic string" for `SystemTabNames.PageHeader` is `"EPiServerCMS_SettingsPanel"`.

3. Start the **AlloyTraining** website, and log in as a CMS admin.
4. Edit the **Alloy Meet Customer Testimonial** teaser, switch to **All Properties** view, and note the the three properties are now in the basic information area, as shown in the following screenshot:



- For your own block types, consider moving the two or three most commonly used properties to the basic information area.

Exercise D5 – Using a block as a content property type

In this exercise, you will define and use an employee block type as a property type.

Prerequisites: complete Exercises B1 to B4.

Using a block as a property type

1. In **AlloyTraining**, in the **Models\Blocks** folder, add an **EmployeeBlock.cs** block type for storing information about employees: **FirstName**, **LastName**, and **HireDate**, as shown in the following code:

```
using EPiServer.Core; // BlockData
using EPiServer.DataAbstraction; // SystemTabNames
using EPiServer.DataAnnotations; // [ContentType]
using System; // DateTime
using System.ComponentModel.DataAnnotations; // [Display]

namespace AlloyTraining.Models.Blocks
{
  [ContentType(DisplayName = "Employee",
    GroupName = SiteGroupNames.Specialized, Order = 10,
    Description = "Use this to store information about an employee.")]
  [SiteBlockIcon]
  public class EmployeeBlock : BlockData
  {
    [Display(Name = "First name",
      GroupName = SystemTabNames.Content, Order = 10)]
    public virtual string FirstName { get; set; }

    [Display(Name = "Last name",
      GroupName = SystemTabNames.Content, Order = 20)]
    public virtual string LastName { get; set; }

    [Display(Name = "Hire date",
      GroupName = SystemTabNames.Content, Order = 30)]
    public virtual DateTime? HireDate { get; set; }
  }
}
```

3. In **AlloyTraining**, in **Views\Shared**, add a new empty Razor view named **EmployeeBlock.cshtml**.
4. Modify the view to use **EmployeeBlock** as its model and to output the **FirstName**, **LastName**, and **HireDate** properties, as shown in the following markup:

```
@model AlloyTraining.Models.Blocks.EmployeeBlock
<div class="block span">
    @Model.FirstName @Model.LastName
    @(Model.HireDate.HasValue ? "was hired on " +
        Model.HireDate.Value.ToString("dddd, d MMMM yyyy") : "")
</div>
```

5. In **AlloyTraining**, open **Models\Pages\StandardPage.cs**.
6. Import the blocks namespace, as shown in the following code:

```
using AlloyTraining.Models.Blocks;
```

7. Add a property, as shown in the following code:

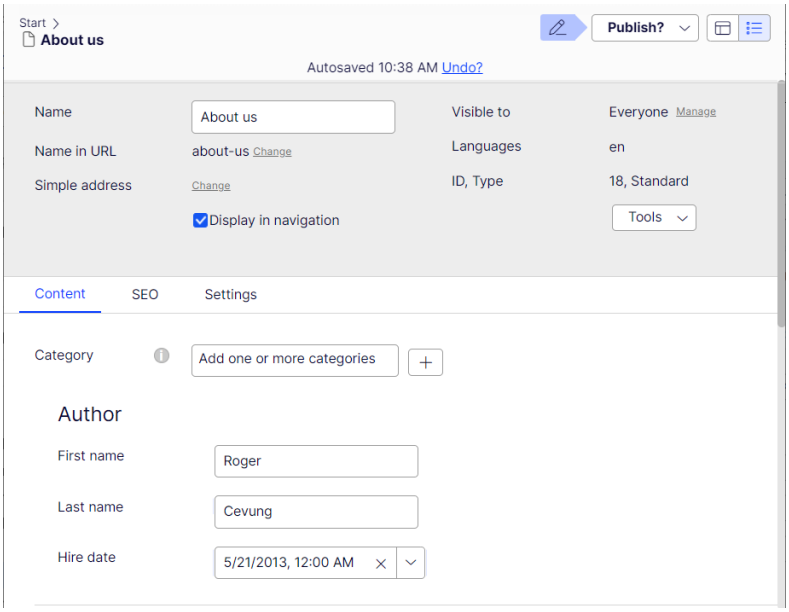
```
public virtual EmployeeBlock Author { get; set; }
```

8. In **AlloyTraining**, open **Views\StandardPage\Index.cshtml**.
9. At the bottom of the view, render the **Author** property with support for on-page editing:

```
@Html.PropertyFor(m => m.CurrentPage.Author)
```

Testing the block type property

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Edit the **About us** page, switch to **All Properties** view, and click the **Content** tab.
3. Modify the **Author** property's three properties, as shown in the following screenshot:



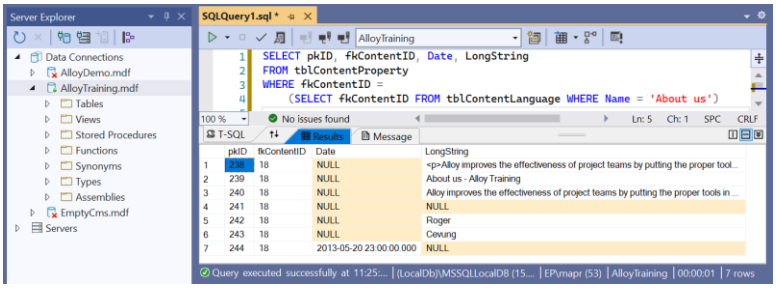
4. **Publish** the page.
5. Switch to **Live** view and note the author's details are rendered onto the page.
6. Close the browser and shut down the web server.

Viewing the property values stored in the database

1. In **App_Data**, double-click **AlloyTraining.mdf** to open a connection to the CMS database in **Server Explorer**.
2. Right-click the database connection, choose **New Query**, and enter a query as shown in the following SQL:

```
SELECT pkID, fkContentID, Date, LongString
FROM tblContentProperty
WHERE fkContentID =
    (SELECT fkContentID FROM tblContentLanguage WHERE Name = 'About us')
```

3. Click the **Execute** button, or press *Ctrl* + *Shift* +*E*, and note the results show that the **Author** property is stored as three separate rows in the table all belonging to the **About us** page with an **fkContentID** of **18**, as shown in the following screenshot:



- The **fkContentID** of your **About us** page might be different!

4. Close the query and do not save changes.
5. Close the database connection.

Module E – Navigating Content

Goal

The overall goal of the exercises in this module is to implement various ways for a visitor to navigate a website. You will:

1. Create a child page listing block.
2. Use the child page listing block in a page for navigating news articles.
3. Create a submenu for navigation.
4. Create a search page for visitors and implement it using Search or Search & Navigation (formerly Find).
5. Add a search box to the top navigation menu.

Exercise E1 – Creating a page listing block

In this exercise, you will create a new block type containing a listing of child pages.

The block will have two properties; a heading and a page picker to select a parent page to show a list of its children.

Prerequisites: complete Exercises B1 to B4.

- Although **Exercise D3 – Creating a preview renderer for partial pages and blocks** is not a prerequisite, if you have not completed that exercise then your blocks will not have On-Page Editing (OPE) experience.

Creating a page listing block type

1. In **AlloyTraining**, in **Models\Blocks**, add a new class named **ListingBlock.cs**.
2. Decorate the class with [ContentType], change the **DisplayName** to **Listing**, add a **Description** of "Choose a page in the tree, and its children will be listed, with a heading.", apply the **[SiteBlockIcon]** attribute to the class, and add the following properties with appropriate attributes:
    a. **Heading**: `string`
    b. **ShowChildrenOfThisPage**: `PageReference`

Your code should look something like the following:

```
using EPiServer.Core; // BlockData, PageReference
using EPiServer.DataAnnotations; // [ContentType]
using System.ComponentModel.DataAnnotations; // [Display]

namespace AlloyTraining.Models.Blocks
{
    [ContentType(DisplayName = "Listing",
        GroupName = SiteGroupNames.Common,
        Description = "Choose a page in the tree, and its children will be listed, with a heading.")]
    [SiteBlockIcon]
    public class ListingBlock : BlockData
    {
        [Display(Name = "Heading", Order = 10)]
        public virtual string Heading { get; set; }

        [Display(Name = "Show children of this page", Order = 20)]
        public virtual PageReference ShowChildrenOfThisPage { get; set; }
    }
```

```
}
```

Creating a page listing view model, controller, and view

1. In **AlloyTraining**, in **Models\ViewModels**, add a new class file named **ListingBlockViewModel.cs**.
2. Add two properties: **Heading** and **Pages**, as shown in the following code:

```
using EPiServer.Core;
using System.Collections.Generic;

namespace AlloyTraining.Models.ViewModels
{
    public class ListingBlockViewModel
    {
        public string Heading { get; set; }
        public IEnumerable<PageData> Pages { get; set; }
    }
}
```

3. In **AlloyTraining**, in **Components**, add a new class named **ListingBlockComponent.cs**.
4. Modify the class to create an instance of the listing block view model and set its properties, before passing it to a partial view, as shown in the following code:

```
using AlloyTraining.Models.Blocks; // ListingBlock
using AlloyTraining.Models.ViewModels; // ListingBlockViewModel
using EPiServer; // IContentLoader
using EPiServer.Core; // PageData, IContent
using EPiServer.Filters; // FilterForVisitor
using EPiServer.Web.Mvc; // BlockComponent
using Microsoft.AspNetCore.Mvc; // IViewComponentResult
using System.Collections.Generic; // IEnumerable<T>
using System.Linq; // Cast<T>

namespace AlloyTraining.Controllers
{
   public class ListingBlockComponent : BlockComponent<ListingBlock>
   {
      protected readonly IContentLoader loader;

      public ListingBlockComponent(IContentLoader loader)
      {
         this.loader = loader;
      }

      protected override IViewComponentResult InvokeComponent(
         ListingBlock currentBlock)
      {
         ListingBlockViewModel viewmodel = new()
         {
            Heading = currentBlock.Heading
         };

         if (currentBlock.ShowChildrenOfThisPage != null)
         {
            IEnumerable<PageData> children = loader.GetChildren<PageData>(
               currentBlock.ShowChildrenOfThisPage);
```

```
            // Remove pages:
            // 1. that are not published
            // 2. that the visitor does not have Read access to
            // 3. that do not have a page template
            IEnumerable<IContent> filteredChildren =
                FilterForVisitor.Filter(children);

            // 4. that do not have "Display in navigation" selected
            viewmodel.Pages = filteredChildren.Cast<PageData>()
                .Where(page => page.VisibleInMenu);
        }

        return View(viewmodel);
    }
  }
}
```

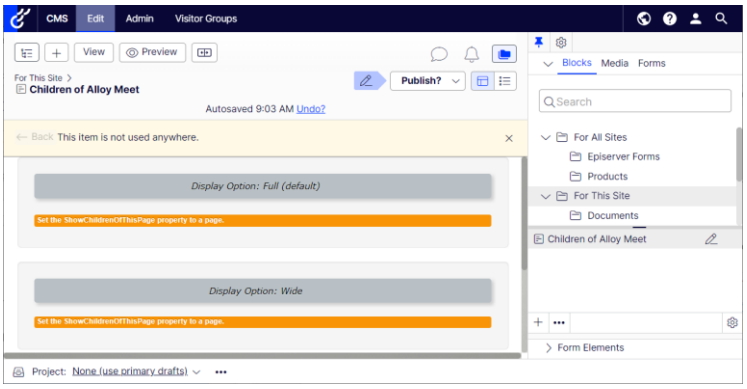
5. In **AlloyTraining**, in **Views\Shared\Components**, add a new folder named **ListingBlock**.
6. In the **ListingBlock** folder, add a new empty Razor view named **Default.cshtml**.
7. Modify the view, as shown in the following markup, and note the following:
    - When there are no pages to show, we render alert messages visible to content editors.
    - `PageData` does not have a strongly-typed `MainBody` property, but the page it references might, so we can use the `Property` property to check if the current page in the listing has a `MainBody` and render it if it does.

```
@using EPiServer.Core
@inject IContextModeResolver modeResolver
@model AlloyTraining.Models.ViewModels.ListingBlockViewModel
@if (Model.Pages == null)
{
  if (modeResolver.CurrentMode == ContextMode.Edit)
  {
    <div class="label label-warning">Set the ShowChildrenOfThisPage property to a page.</div>
  }
}
else
{
  <h2 @Html.EditAttributes(x => x.Heading)>@Model.Heading</h2>
  if (Model.Pages.Count() == 0)
  {
    <div class="label label-warning">The page selected has no children.</div>
  }
  foreach (PageData page in Model.Pages)
  {
    <div class="listresult theme1">
      <h3>@Html.ContentLink(page.ContentLink)</h3>
      @if (page.StartPublish.HasValue)
      {
        <p class="date">Published on
          @(page.StartPublish.Value.ToString("dddd, d MMMM yyyy"))
        </p>
      }
      @if (page.Property["MainBody"] != null)
      {
```

```
            @Html.Raw(page.Property["MainBody"].Value)
        }
        <hr />
    </div>
  }
}
```

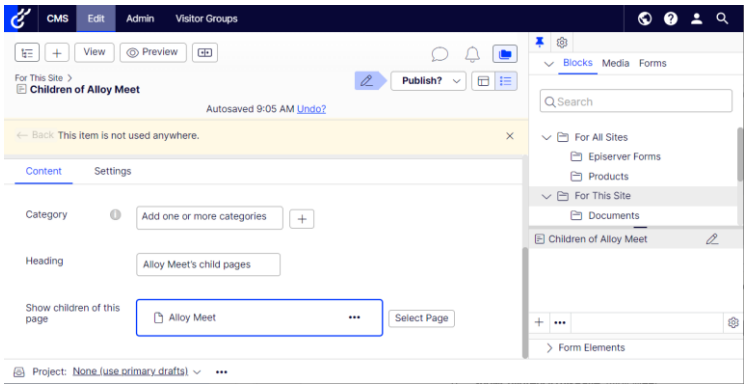
Testing the page listing block

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. In **Assets** pane, click **Blocks**.
3. Add a new **Listing** block to the **For This Site** folder named **Children of Alloy Meet**.
4. Switch to **On-Page Editing** view, and note the warning to editors, as shown in the following screenshot:

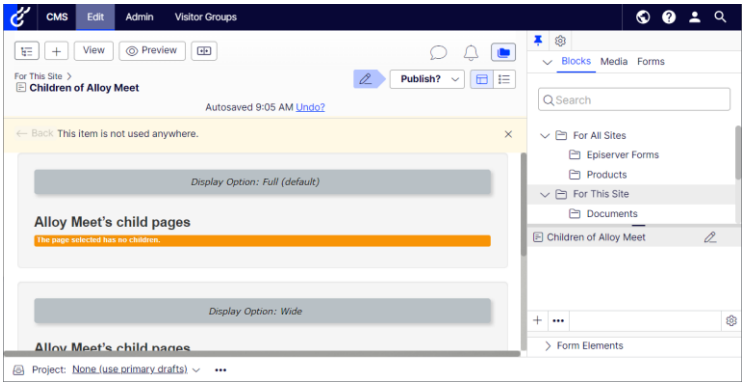


- You must have completed **Exercise D3 – Creating a preview renderer for partial pages and shared blocks** on page 166 to have the On-Page Editing experience.

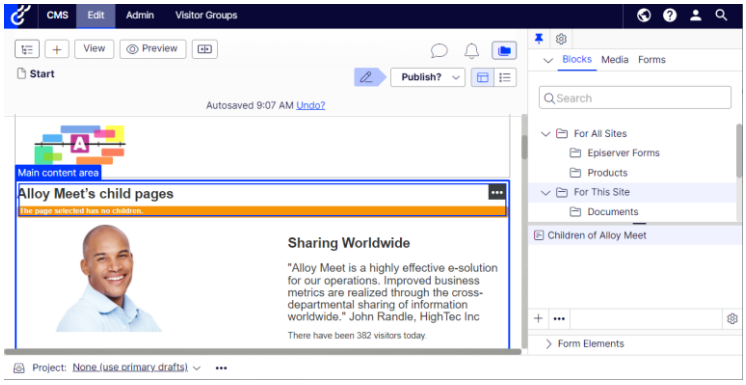
5. Switch to **All Properties** view, and set the properties, as shown in the screenshot:
    a. **Heading**: Alloy Meet's child pages
    b. **ShowChildrenOfThisPage**: Alloy Meet



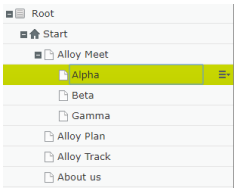
6. Switch to **On-Page Editing** view, and note the warning to editors, as shown in the following screenshot:



7. **Publish** the block.
8. Edit the **Start** page, and drag and drop **Children of Alloy Meet** into the top of its main content area, as shown in the following screenshot:



9. **Publish** the page.
10. Add some standard pages underneath **Alloy Meet** in the page tree, named **Alpha**, **Beta**, and **Gamma**, as shown in the screenshot:

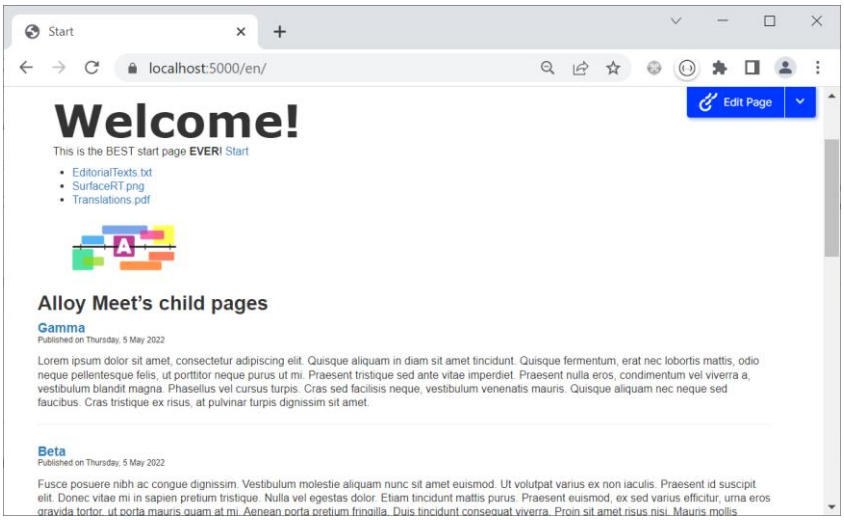


11. For each page, set their **Main body** properties using a Lorem Ipsum generator.

- Lorem Ipsum is dummy text of the printing and typesetting industry. Learn about and generate some at the following link: https://www.lipsum.com/

12. Switch to **Live** view.

13. View the **Start** page, and note the output, as shown in the following screenshot:



14. Close the browser and shut down the web server.

- You have now implemented a simple listing block. In the next exercise, you will use it to implement a news page that lists articles.

Exercise E2 – Creating a news landing page

In this exercise, you will create a news landing page type that lists news articles beneath it in the page tree.

You will use the standard page type as a base for the news landing page, you will use the page listing block from the previous exercise to add a news listing function to the news landing page, and you will create instances of the standard page under the news landing page to be its child news articles.

Prerequisites: complete Exercises B1 – B4, E1.

Creating a news landing page type

1. In **AlloyTraining**, in **Models\Pages**, add a new class file named **NewsLandingPage.cs**.
2. Modify the class to inherit from **StandardPage**, decorate the class with [ContentType], change the **DisplayName** to **News Landing**, add a **Description** of "Use this as a landing page for a list of news articles.", put the page in the **News** group, apply the **[SitePageIcon]** attribute to the class, and add a **ListingBlock** property named **NewsListing** with appropriate attributes.

- It would be nice if we could override the `SetDefaultValues` method and set the `NewsListing` property's `Heading` to the `Name` of the page and the `Parent` to reference this page, i.e. the news landing page will show a listing of its child pages, but unfortunately, the `Name` and `PageLink` properties are empty during `SetDefaultValues`.

Your code should look something like the following:

```
using AlloyTraining.Models.Blocks; // ListingBlock
using EPiServer.DataAnnotations; // [ContentType]
using System.ComponentModel.DataAnnotations; // [Display]

namespace AlloyTraining.Models.Pages
{
    [ContentType(DisplayName = "News Landing",
        GroupName = SiteGroupNames.News,
        Description = "Use this as a landing page for a list of news articles.")]
    [SitePageIcon]
    public class NewsLandingPage : StandardPage
    {
        [Display(Name = "News listing", Order = 315)]
        public virtual ListingBlock NewsListing { get; set; }
    }
}
```

Creating a news landing page template

1. In **AlloyTraining**, in **Controllers**, add a new class file named **NewsLandingPageController.cs**.
2. Modify the class to derive from **PageControllerBase<T>**, and the **Index** action method to use a view model, as shown in the following code:

```
using AlloyTraining.Models.Pages; // NewsLandingPage
using EPiServer; // IContentLoader
using Microsoft.AspNetCore.Mvc; // IActionResult

namespace AlloyTraining.Controllers
{
    public class NewsLandingPageController
        : PageControllerBase<NewsLandingPage>
    {
```

```
        public NewsLandingPageController(
            IContentLoader loader) : base(loader)
        {
        }

        public IActionResult Index(NewsLandingPage currentPage)
        {
            return View(CreatePageViewModel(currentPage));
        }
    }
}
```

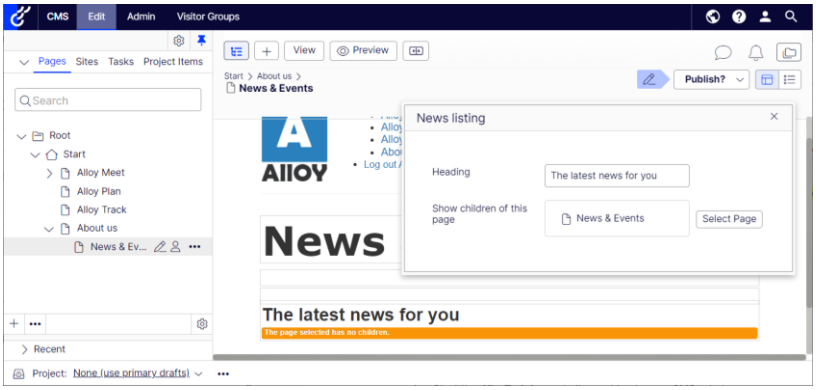
3. In **AlloyTraining**, in **Views**, add a new folder named **NewsLandingPage**.
4. In **NewsLandingPage**, add a new empty Razor file named **Index.cshtml**.
5. Modify the view, as shown in the following markup:

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model PageViewModel<NewsLandingPage>
@{
    Layout = "~/Views/Shared/Layouts/_LeftNavigation.cshtml";
}
<h1 @Html.EditAttributes(m => m.CurrentPage.MetaTitle)>
    @(Model.CurrentPage.MetaTitle ?? Model.CurrentPage.Name)
</h1>
<p class="introduction">
    @Html.PropertyFor(m => m.CurrentPage.MetaDescription)
</p>
<div class="row">
    <div class="span8">
        @Html.PropertyFor(m => m.CurrentPage.MainBody)
        @Html.PropertyFor(m => m.CurrentPage.NewsListing)
    </div>
</div>
```

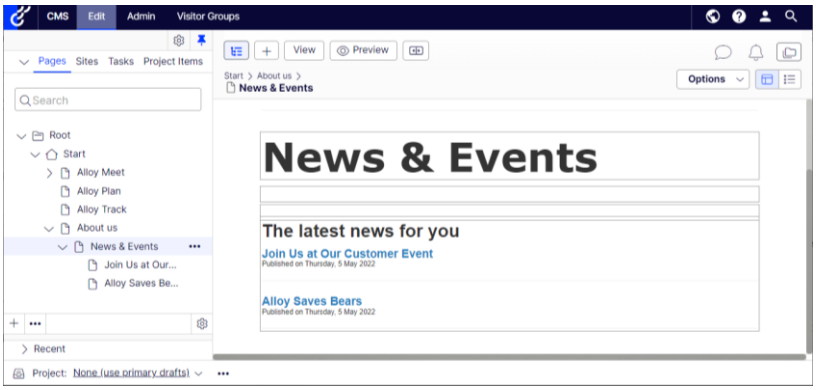
Testing the news landing page

1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Under **About us**, add a new **News Landing** page named **News & Events**.
3. In **On-Page Editing** view, click the **News listing** property.

4. Set **Heading** to **The latest news for you**, and **Show children of this page** to reference the **News & Events** page, as shown in the following screenshot:



5. **Publish** the page.
6. Under **News & Events**, add two **Standard** pages named **Alloy Saves Bears** and **Join Us at Our Customer Event**, and publish them.
7. Edit **News & Events**, and note the two news articles listed on the page, as shown in the following screenshot:



8. Close the browser and shut down the web server.

Exercise E3 – Improving navigation menus

In this exercise, you will replace the simple menu from an earlier exercise with a better-looking menu using an extension method named MenuList.

The basics of what this **MenuList** helper does is that it allows you to define a markup template for each item in your menu and it does filtering to remove unwanted items (for example, pages with no renderer can be filtered out).

Prerequisites: complete Exercises B1 to B4.

Improving the top navigation menu

1. In **AlloyTraining**, open **Views\Shared_NavigationMenu.cshtml**.
2. Modify the view, as shown in the following markup:

```
@using EPiServer.Core
@using AlloyTraining.Business.ExtensionMethods
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@model IPageViewModel<SitePageData>
@{
  HelperResult ItemTemplate(HtmlHelpers.MenuItem item)
  {
    <li class="nav-item">
      @Html.PageLink(item.Page, null, new { @class = "nav-link" + (item.Selected ? " active" : null) })
    </li>
    return new HelperResult(w => Task.CompletedTask);
  }
}
<div class="alloyMenu">
  <div class="navbar">
    <div class="navbar-inner">
      <div class="container">
        <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </a>
        <div class="nav-collapse">
          <ul class="nav">
            <li
class="@(Model.CurrentPage.ContentLink.CompareToIgnoreWorkID(ContentReference.StartPage) ? "active"
: null)">
              @Html.ContentLink(ContentReference.StartPage)
            </li>
            @Html.MenuList(ContentReference.StartPage, ItemTemplate)
            <li>
              @if (User.Identity.IsAuthenticated)
              {
```

```
                    <a href="logout">Log out @User.Identity.Name</a>
                }
                else
                {
        <a href="/util/login?ReturnUrl=
        @Model.CurrentPage.PageLink.ExternalURLFromReference()">Log in</a>                    }
                </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
</div>
```

- To save time, and avoid line breaks, drag and drop or copy and paste the file from the exercise files ZIP.

Adding a left navigation submenu

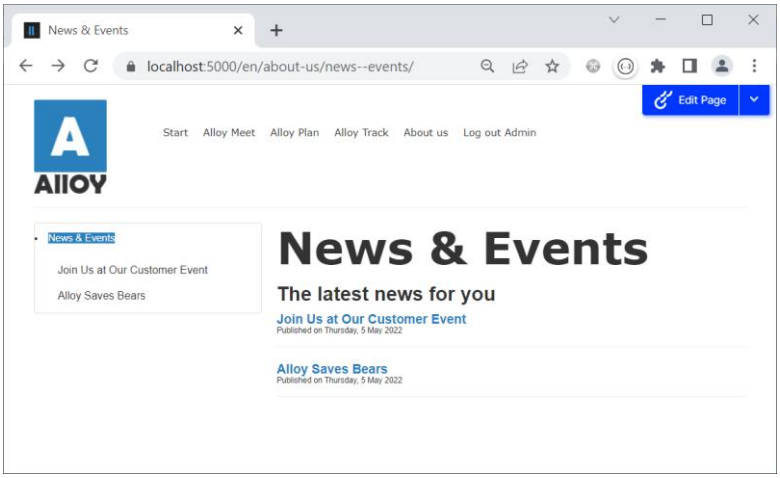
1. In **AlloyTraining**, in **Views\Shared**, add an empty Razor view named **_LeftNavigationMenu.cshtml**.
2. Modify the view, as shown in the following markup:

```
@using AlloyTraining.Models.ViewModels
@using AlloyTraining.Models.Pages
@using AlloyTraining.Business.ExtensionMethods
@model IPageViewModel<SitePageData>
@{
  HelperResult ItemTemplate(HtmlHelpers.MenuItem item)
  {
    <li class="nav-item">
      <a href="@Url.ContentUrl(item.Page.PageLink)"
        class="nav-link @(item.Selected ? " active" : null)">
        @item.Page.PageName
      </a>
      @if (item.Selected)
      {
        <ul class="nav nav-pills flex-column">
          @Html.MenuList(item.Page.ContentLink, ItemTemplate)
        </ul>
      }
    </li>
    return new HelperResult(w => Task.CompletedTask);
  }
}
<div id="alloyDrop" class="accordion">
  <div class="accordion-group">
    @if (Model.Section != null)
    {
      @Html.MenuList(Model.Section.ContentLink, ItemTemplate)
    }
  </div>
</div>
```

3. In **Views\Shared\Layouts_LeftNavigation.cshtml**, add a statement to render the **_LeftNavigationMenu**, inside `<div class="span4">`, as shown in the following markup:

```
@await Html.PartialAsync("_LeftNavigationMenu", Model)
```

4. Start the **AlloyTraining** website, log in, and navigate to **About us** | **News & Events**, and note the top and left navigation menus, as shown in the following screenshot:



5. Close the browser and shut down the web server.

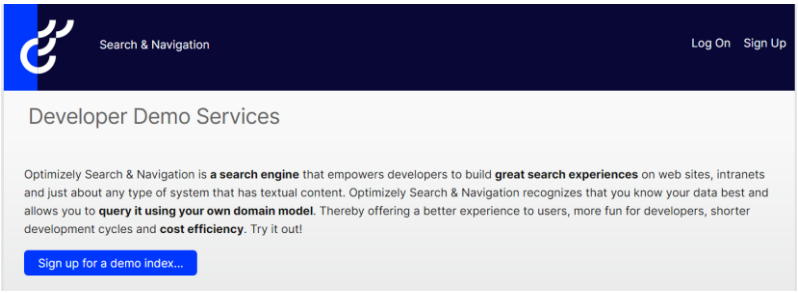
Exercise E4 – Creating a search page for visitors

In this exercise, you will add a search page to the AlloyTraining site, allowing visitors to perform free-text searches on pages using Search & Navigation (formerly Find).

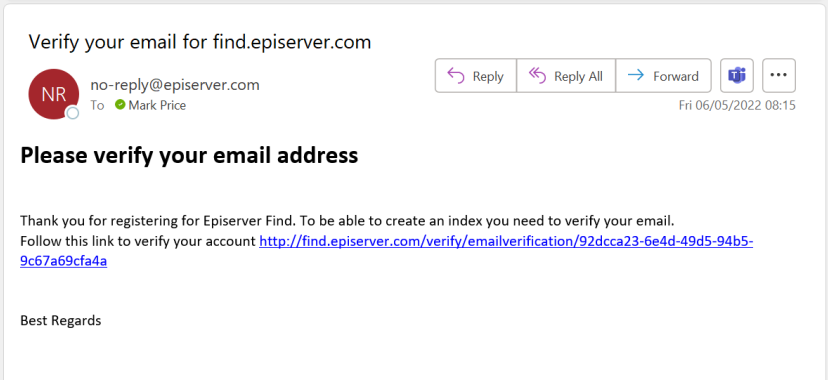
Prerequisites: complete Exercises B1 to B4.

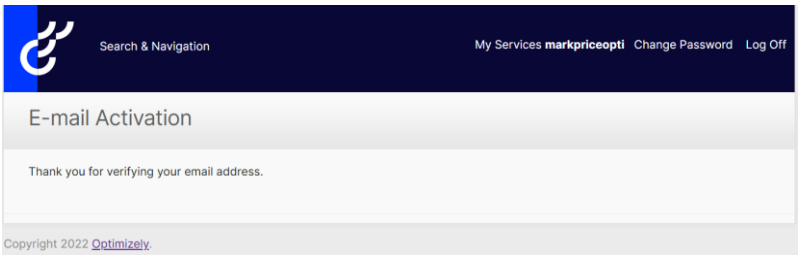
Signing up for a free Search & Navigation index

1. Open a web browser and navigate to https://find.episerver.com/



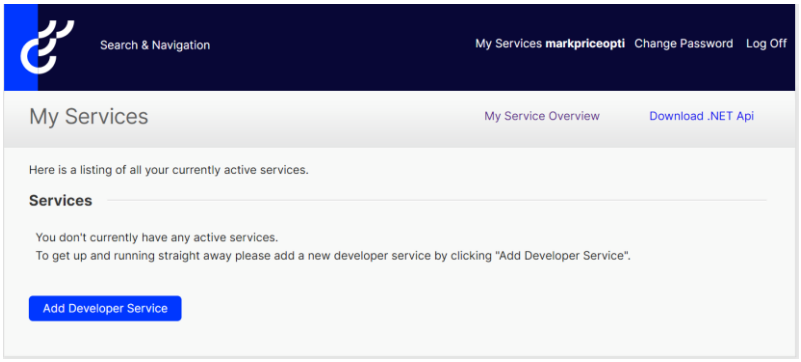
2. Click **Sign up for a demo index...** or **Sign Up** in the top right corner.
3. Fill in the required information and click **Register**.
4. When the verification email arrives, copy and paste the link into your browser's address box and press ENTER to confirm your address and activate the index.



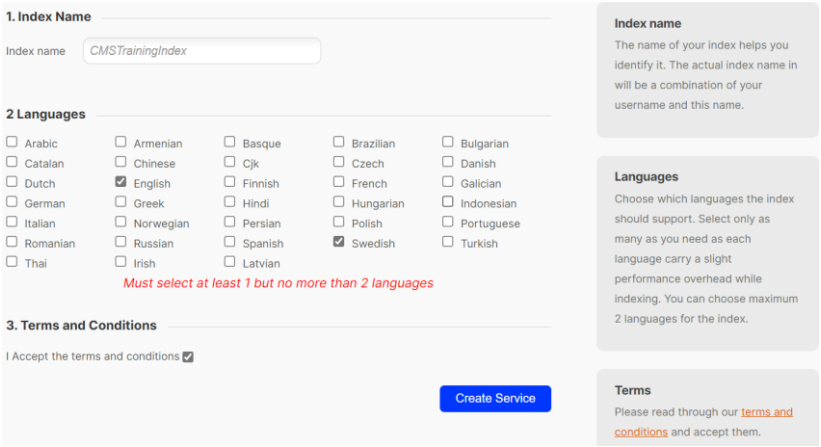


- If you don't receive your verification email, check your junk mail folder!

5. On the **E-mail Activation** verification page, click **My Services**.
6. In **My Services**, click **Add Developer Service**, as shown in the following screenshot:

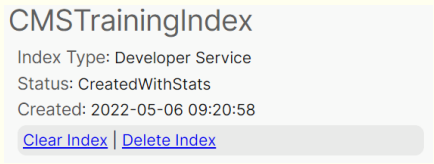


7. Fill in a name for your index of less than 18 characters. The name is technically irrelevant but should preferably represent what you're using the index for, such as, **CMSTrainingIndex**.
8. Select **English** and **Swedish** from the list of languages, check the terms and conditions checkbox, and then click **Create Service**, as shown in the following screenshot:



9. You should now see a list of details about your index. Note the **Status**: it will probably be **NotCreated**, as shown in the following screenshot:

- If you wait a minute and refresh the page, it should say **CreatedWithStats**, as shown in the following screenshot:



10. Note the **Configuration** is for our older platforms as it would be copied to **Web.config**, as shown in the following screenshot:

- In the next section, you will create the equivalent in **appsettings.json** for modern .NET platforms.

Installing and configuring Search & Navigation

1. Open your project.
2. Navigate to **Tools** | **NuGet Package Manager** | **Package Manager Console**.
3. Enter the following command to install the CMS integration package for Search & Navigation:

```
Install-Package EPiServer.Find.Cms
```

4. Confirm that you have the Search & Navigation package referenced, as shown in the following JSON:

```
<ItemGroup>
  <PackageReference Include="EPiServer.CMS" Version="12.9.0" />
  <PackageReference Include="EPiServer.Find.Cms" Version="14.1.0" />
  <PackageReference Include="Wangkanai.Detection" Version="5.2.0" />
</ItemGroup>
```

5. Build the project.
6. In **appsettings.Development.json**, copy the database connection section, and paste it into **appsettings.json**, as shown highlighted in the following configuration:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "Microsoft": "Warning",
      "EPiServer": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
```

```
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "EPiServerDB": "Data
Source=(LocalDb)\\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\\AlloyTraining.mdf;Initial
Catalog=AlloyTraining;Integrated Security=True;Connect Timeout=30"
  }
}
```

7. Replace **|DataDirectory|** with the absolute path, as shown highlighted in the following configuration:

```
"EPiServerDB": "Data
Source=(LocalDb)\\MSSQLLocalDB;AttachDbFilename=C:\\Optimizely\\CmsTraining\\AlloyTraining\
\App_Data\\AlloyTraining.mdf;Initial Catalog=AlloyTraining;Integrated Security=True;Connect
Timeout=30"
```

8. Enter the following command to update the database schema:

```
dotnet-episerver update-database AlloyTraining\AlloyTraining.csproj
```

- If necessary, replace **AlloyTraining** with **AlloyDemo** or whatever your project name is.

9. Note the results, as shown in the following edited output:

```
---------------------------------------
- Optimizely database cli tool, v2.0.0 -
---------------------------------------

[08:36:42 INF] Running update command.
[08:36:57 INF] Processing
C:\Users\mapr\.nuget\packages\EPiServer.Find.Cms\14.1.0\tools\epiupdates\sql\1.0.1.sql
[08:36:57 INF] Script validation: 1 - Upgrading database
[08:36:57 INF] Running script 1.0.1.sql
[08:36:57 INF] Processing
C:\Users\mapr\.nuget\packages\EPiServer.CMS.Core\12.3.0\tools\epiupdates\sql\7.8.0.sql
[08:36:57 INF] Script validation: 0 - Already correct database version
...
[08:36:58 INF] Processing
C:\Users\mapr\.nuget\packages\EPiServer.Find.Cms\14.1.0\tools\epiupdates\sql\12.2.8.sql
[08:36:58 INF] Script validation: 1 - Upgrading database
[08:36:58 INF] Running script 12.2.8.sql
[08:36:58 INF] Processing
C:\Users\mapr\.nuget\packages\EPiServer.Find.Cms\14.1.0\tools\epiupdates\sql\12.4.2.sql
[08:36:58 INF] Script validation: 1 - Upgrading database
[08:36:58 INF] Running script 12.4.2.sql
...
[08:36:58 INF] Processing
C:\Users\mapr\.nuget\packages\EPiServer.Find.Cms\14.1.0\tools\epiupdates\sql\13.4.1.sql
[08:36:58 INF] Script validation: 1 - Upgrading database
[08:36:58 INF] Running script 13.4.1.sql
```

10. Open **appsettings.json**.
11. Add elements to configure automatic database schema updates in the future and your Search & Navigation index (you must copy and paste the values from your **Configuration** page and the **Private URL** because it contains a private account key), as shown in the following configuration:

```
{
  ...
  "EPiServer": {
    "Cms": {
      "DataAccess": {
        "UpdateDatabaseSchema": "true"
```

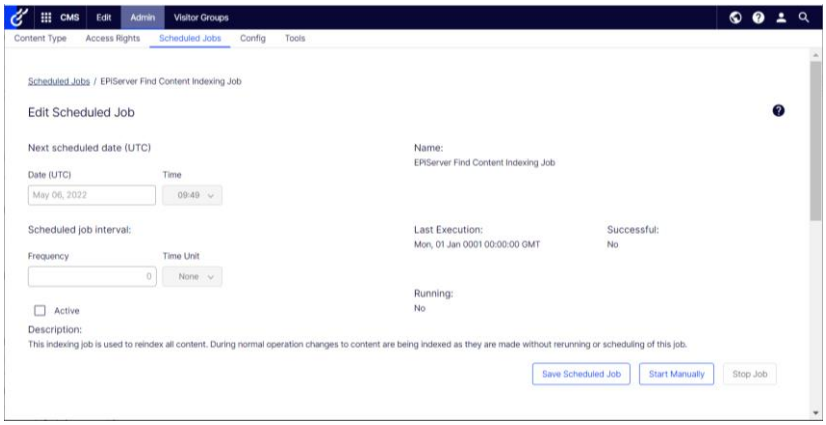
```
    }
  },
  "Find": {
    "ServiceUrl": "https://demo02.find.episerver.net/ohakh...Da4w6",
    "DefaultIndex": "yourname_indexname"
  }
 }
}
```

10. In Startup.cs, in the `ConfigureServices` method, after the call to `AddCms`, add a call to the `AddFind` extension method, as shown highlighted in the following code:

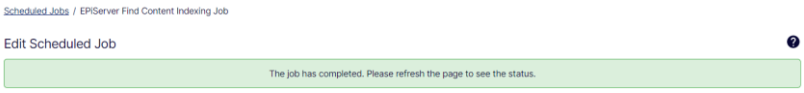
```
services
    .AddCmsAspNetIdentity<ApplicationUser>()
    .AddCms()
    .AddFind() // Search & Navigation
    .AddAdminUserRegistration()
    .AddEmbeddedLocalization<Startup>();
```

Testing the website index

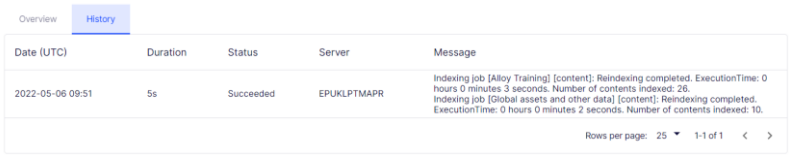
1. Start the **AlloyTraining** site, and log in as a CMS admin.
2. Navigate to **CMS** | **Admin** | **Scheduled Jobs** | **EPiServer Find Content Indexing Job**, as shown in the following screenshot:



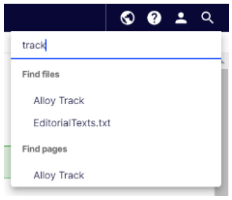
3. Click **Start Manually**, and wait for the job to complete, as shown in the following screenshot:



4. Click the **History** tab, and note the number of content items indexed, as shown in the following screenshot:



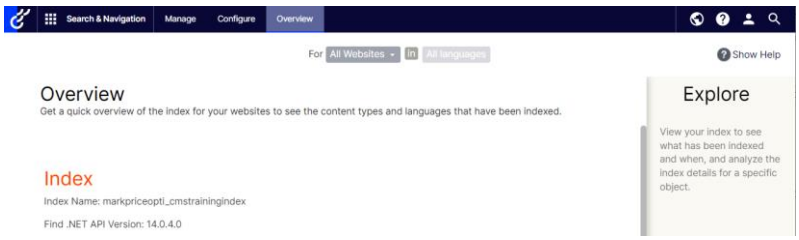
5. In the top menu, click the search icon (magnifying glass), and inside the **Search** box, and enter **track**. You should see multiple matches, with results shown from Search & Navigation (labeled **Find files** and **Find pages**), as shown in the screenshot:



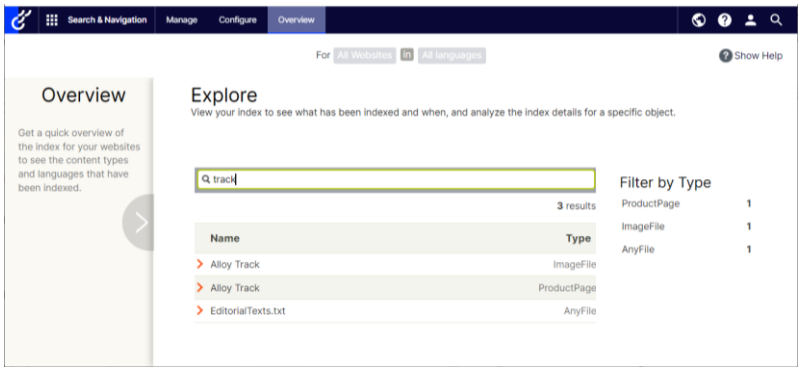
Reviewing the administrator's view

1. In the top menu, click the product selector, as shown as a nine box grid in the following screenshot:

2. Navigate to **Search & Navigation** | **Overview**, as shown in the following screenshot:



3. Click the **Explore** area, enter **track** in the search box, and note the filtered content, as shown in the following screenshot:



4. Close the browser and shut down the web server.

Creating a search page type

- This would be a good example of a page type that would suit the "Feature Folder" idea since the page type is less a type of content and more a feature of the website. But we will keep it simple and treat it like any other page type.

1. In **AlloyTraining**, in **Models\Pages**, add a new class named **SearchPage.cs**.
2. Modify the class to inherit from **StandardPage**, docorate the class with the `[ContentType]` attribute, change the **DisplayName** to **Search**, group the page type under **Specialized**, add a **Description** of "Use this to enable visitors to search for pages and media on the site.", and apply the **[SiteSearchIcon]** attribute to the class.

Your code should look something like the following:

```
using EPiServer.DataAnnotations; // [ContentType]

namespace AlloyTraining.Models.Pages
{
   [ContentType(DisplayName = "Search",
      GroupName = SiteGroupNames.Specialized, Order = 30,
      Description = "Use this to enable visitors to search for pages and media on the site.")]
   [SiteSearchIcon]
   public class SearchPage : StandardPage
   {
   }
}
```

Creating a search page view model

1. In **AlloyTraining**, in **Models\ViewModels**, add a new class named **SearchPageViewModel.cs**.
2. Inherit from `PageViewModel<SearchPage>` and modify the contents, to add some properties for the visitor's free-text search term and for the results, as shown in the following code:

```
using AlloyTraining.Models.Pages; // SearchPage
using System.Collections.Generic; // List<T>

namespace AlloyTraining.Models.ViewModels
{
   public class SearchPageViewModel : PageViewModel<SearchPage>
   {
      public SearchPageViewModel(
         SearchPage currentPage) : base(currentPage)
      {
      }

      public string SearchText { get; set; }
      public List<Result> SearchResults { get; set; }
   }

   public class Result
   {
      public string Title { get; set; }
      public string Description { get; set; }
      public string Url { get; set; }
   }
}
```

- We need to declare a class to represent each result because we want to keep search results abstract and not tied to a specific technology like **Search & Navigation** or the older Lucene-based **Search**.

Implementing a search page controller using Search & Navigation

1. In **AlloyTraining**, in **Controllers**, add a new class named **SearchPageController.cs**.
2. Modify the class to inherit from **PageControllerBase<T>**, create an instance of the search page view model and set its properties, before passing it to a view, as shown in the following code:

```
using AlloyTraining.Business.ExtensionMethods; // GetSection()
using AlloyTraining.Models.Pages; // SearchPage, StartPage, SitePageData
using AlloyTraining.Models.ViewModels; // SearchPageViewModel
using EPiServer; // IContentLoader
using EPiServer.Core; // ContentReference
using EPiServer.Filters; // FilterForVisitor
using EPiServer.Find; // For()
using EPiServer.Find.Cms; // FilterForVisitor()
using EPiServer.Find.Framework; // SearchClient
using Microsoft.AspNetCore.Mvc; // IActionResult
using System.Linq; // Cast<T>(), Select()

namespace AlloyTraining.Controllers
{
   public class SearchPageController : PageControllerBase<SearchPage>
   {
      public SearchPageController(IContentLoader loader) : base(loader)
      {
      }

      public IActionResult Index(SearchPage currentPage, string q)
      {
         SearchPageViewModel viewmodel = new(currentPage);

         viewmodel.StartPage =
            loader.Get<StartPage>(ContentReference.StartPage);

         viewmodel.MenuPages = FilterForVisitor.Filter(
            loader.GetChildren<SitePageData>(ContentReference.StartPage))
            .Cast<SitePageData>().Where(page => page.VisibleInMenu);

         viewmodel.Section = currentPage.ContentLink.GetSection();

         if (!string.IsNullOrWhiteSpace(q))
         {
            viewmodel.SearchText = q;

            ITypeSearch<SitePageData> query = SearchClient.Instance
               .Search<SitePageData>() // 1. only site pages
               .For(q)                 // 2. free-text query
               .FilterForVisitor()     // 3. only what the visitor can read
               .FilterOnCurrentSite(); // 4. only under the Start page
                                       //    (e.g. to exclude Wastebasket)

            IContentResult<SitePageData> results = query.GetContentResult();
```

```
            viewmodel.SearchResults = results
                .Select(x => new Result
                {
                    Title = x.MetaTitle ?? x.Name,
                    Description = x.MetaDescription?.TruncateAtWord(20),
                    Url = x.PageLink.ExternalURLFromReference()
                }).ToList();
        }

        return View(viewmodel);
    }
  }
}
```

Creating a search page view

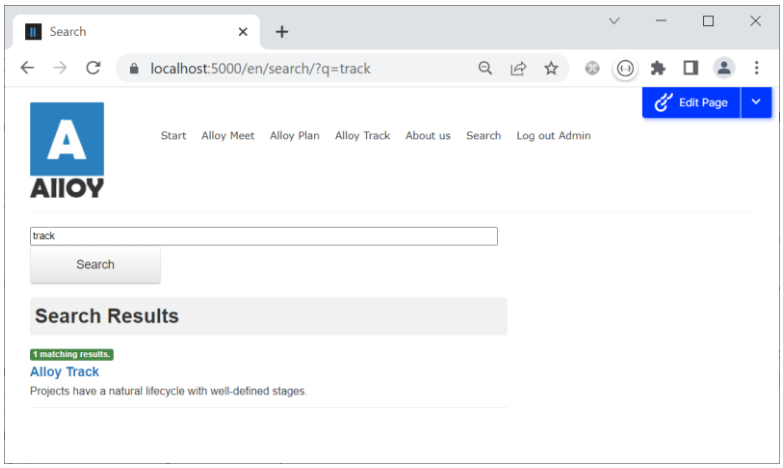
1. In **AlloyTraining**, in **Views**, add a new folder named **SearchPage**.
2. In the **SearchPage** folder, add a new empty Razor view named **Index.cshtml**.
3. Modify the view, as shown in the following markup, and note the following:
    a. To support searching in Edit view, the form must `POST` instead of `GET`.
    b. If there is search text, the Search Results are shown, and then either shows a warning if there are no matches, or the total hits and the list of results.

```
@using AlloyTraining.Models.ViewModels
@inject IContextModeResolver modeResolver
@model SearchPageViewModel
<div class="row">
  <div class="span8">
    @using (Html.BeginForm(actionName: null,
     controllerName: null, routeValues: null,
     method: modeResolver.CurrentMode ==
       ContextMode.Edit ? FormMethod.Post : FormMethod.Get))
    {
      <input tabindex="1" name="q" value="@Model.SearchText" />
      <input type="submit" tabindex="2" class="btn" value="Search" />
    }
    @if (!string.IsNullOrWhiteSpace(Model.SearchText))
    {
      <div class="row">
        <div class="span8 grayHead">
          <h2>Search Results</h2>
        </div>
      </div>
      if (Model.SearchResults.Count == 0)
      {
        <div class="row">
          <div class="span8 SearchResults">
            <span class="label label-warning">No matching results.</span>
          </div>
        </div>
      }
      else
      {
```

```
            <div class="row">
              <div class="span8 SearchResults">
                <span class="label label-success">
                @Model.SearchResults.Count matching results.</span>
                @foreach (var item in Model.SearchResults)
                {
                  <div class="listResult">
                    <h3><a href="@item.Url">@item.Title</a></h3>
                    <p>@item.Description</p>
                    <hr />
                  </div>
                }
              </div>
            </div>
          }
        }
      </div>
      <div class="span4">
        @Html.PropertyFor(m => m.CurrentPage.MainBody)
      </div>
    </div>
```

Creating and testing the search page

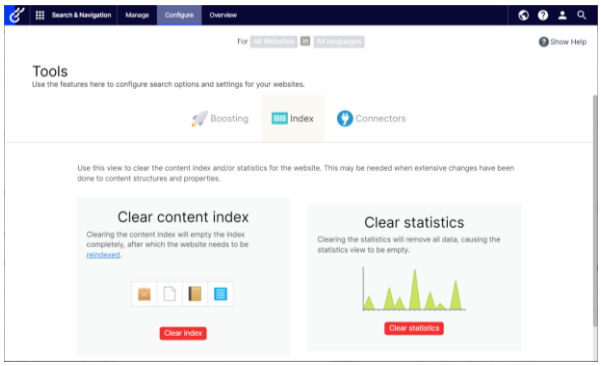
1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Add a new **Search** page named **Search** under the **Start** page.
3. Publish the **Search** page.
4. Switch to **Live view** and click **Search** to navigate to the **Search** page as a visitor.
5. Enter the search text **track**, and press *ENTER* or click **Search**, and note the search results, as shown in the following screenshot:



Resetting the search index for Search & Navigation

1. Navigate to **Search & Navigation** | **Configure** | **Index**.

2. Click **Clear Index**, as shown in the following screenshot:



3. Navigate to **CMS** | **Admin** | **Scheduled Jobs** | **EPiServer Find Content Indexing Job**.
4. Click **Start Manually** and wait for the job to complete.

Exercise E5 – Adding a search box to the top navigation menu

In this exercise, you will implement a search box to use the logic that was added in the Search Page in the previous exercise to perform searches from other places in your site, giving visitors to the site a better experience and more options.

You will add a search field and a button to the existing main navigation control, add a new property to the start page that holds a reference to the Search page and then write logic that, when the search button in the main navigation is clicked, adds the search text to the query string and does a redirect to the search page.

Prerequisites: complete Exercises B1 – B4, E4.

Adding a property to the Start page to reference the Search page

1. In **AlloyTraining**, in **Models\Pages\StartPage.cs**, add a property to reference an instance of the search page type, as shown in the following code:

```
[Display(Name = "Search page",
    Description = "If you add a Search page to the site, set this property to
reference it to enable search from every page.",
    GroupName = SiteTabNames.SiteSettings,
    Order = 40)]
[AllowedTypes(typeof(SearchPage))]
public virtual PageReference SearchPageLink { get; set; }
```

Adding a search box to the navigation menu

1. In **AlloyTraining**, open **Views\Shared_NavigationMenu.cshtml**.
2. At the top of the file, add a statement to inject the IContextModeResolver dependency service, as shown in the following code:

```
@inject IContextModeResolver modeResolver
```

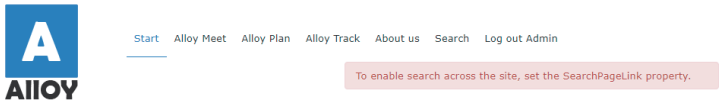
3. After the <ul class="nav"> element, inside the element, add the following markup:

```
<div class="navbar-search pull-right">
  @if (Model.StartPage != null)
  {
    if (PageReference.IsNullOrEmpty(Model.StartPage.SearchPageLink))
    {
      if (modeResolver.CurrentMode == ContextMode.Edit)
      {
        <div class="alert alert-danger">
          To enable search across the site,
          set the SearchPageLink property.
        </div>
      }
    }
    else
    {
      <form action="@Model.StartPage.SearchPageLink.ExternalURLFromReference()"
```

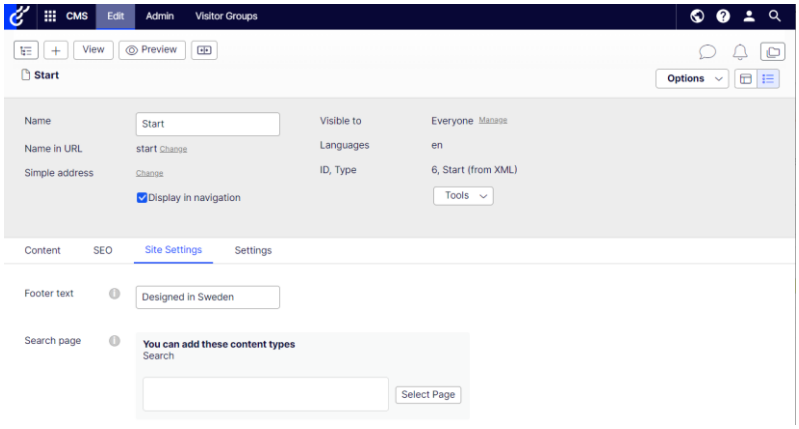
```
        method="post">
            <input type="text" class="search-query" name="q"
            id="SearchKeywords" placeholder="Search" />
            <input type="submit" class="searchButton" id="SearchButton" value="" />
        </form>
    }
  }
</div>
```

Enable the search box and test the website

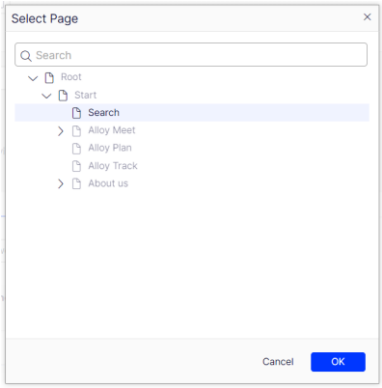
1. Start the **AlloyTraining** website, and log in as a CMS admin.
2. Edit the **Start** page, and note the warning for editors, as shown in the following screenshot:



3. Switch to **All Properties** view, click **Site Settings**, and note that you can set a property for the **Search page**, as shown in the following screenshot:



4. Click the **Select Page** button, and then set the **Search page** property to reference the **Search** page, as shown in the screenshot:



5. Publish the changes to the **Start** page.
6. Switch to **Live view**.
7. Navigate to any page, enter **plan** in the search box, click the search button or press *ENTER*, and note you are directed to the **Search** page to see the results.
8. Close the browser and shut down the web server.

Exercise E6 – Working with Grid View

In this exercise, you will add the grid view labs add-on to a CMS site, and then customize it.

Prerequisites: complete Exercise A1.

Installing and activating the grid view for the AlloyDemo website

1. Open the solution with the **AlloyDemo** project.
   (This is the project that used the **Alloy MVC** project template, *not* the **Empty** template.)
2. In the **AlloyDemo** project, add a package reference to `EPiServer.Labs.GridView`, as shown highlighted in the following markup:

```
<ItemGroup>
  <PackageReference Include="EPiServer.CMS" Version="12.7.0" />
  <PackageReference Include="EPiServer.Labs.GridView" Version="1.0.2" />
  <PackageReference Include="Wangkanai.Detection" Version="5.2.0" />
</ItemGroup>
```

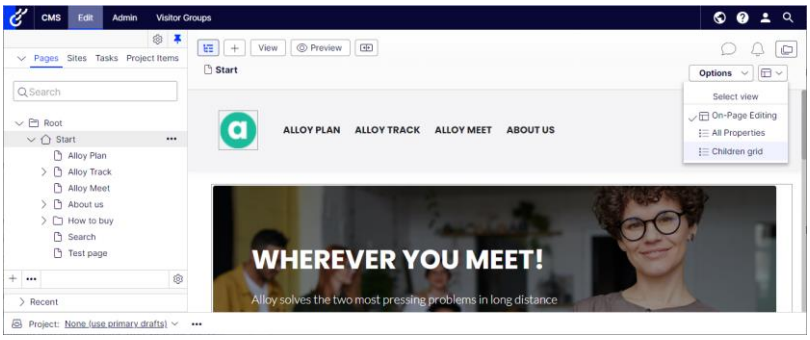
3. In **Startup.cs**, import the namespace for working with grid view, as shown in the following code:

```
using EPiServer.Labs.GridView; // AddGridView extension method
```

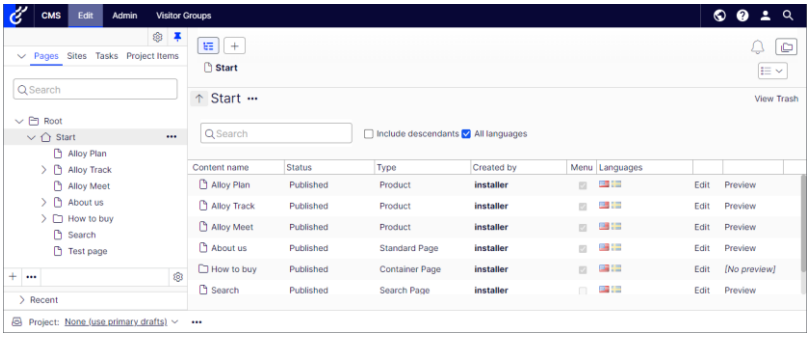
4. In the `ConfigureServices` method, add a statement to enable the grid view aka **Children grid** to be selected as a view as well as **On-Page Editing** and **All Properties**, as shown in the following code:

```
services.AddGridView(options =>
{
   options.IsViewEnabled = true;
});
```

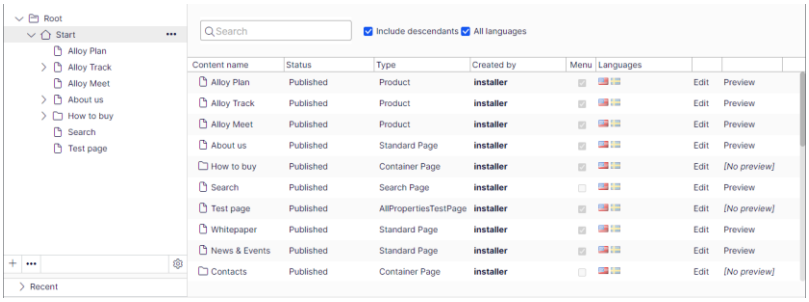
5. Start the **AlloyDemo** website project.
6. Log in as an administrator.
7. In **Edit** view, note the pair of buttons to toggle between **On-Page Editing** and **All Properties** view has changed to a dropdown with three choices, including **Children grid**, as shown in the following screenshot:



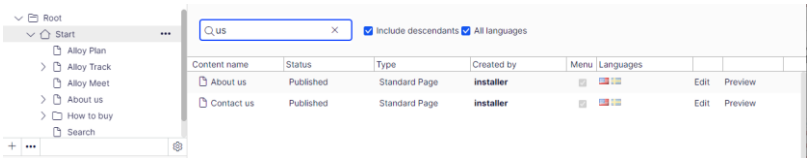
8. Select **Children grid** and note the view shows a grid of common properties for all the children of the **Start** page, as shown in the following screenshot:



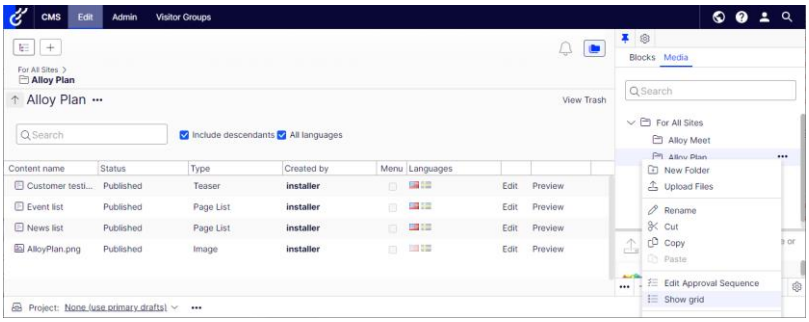
9. Select the **Include descendents** check box and note the grid populates with all pages within the current site because we have effectively flattened the page tree hierarchy, as shown in the following screenshot:



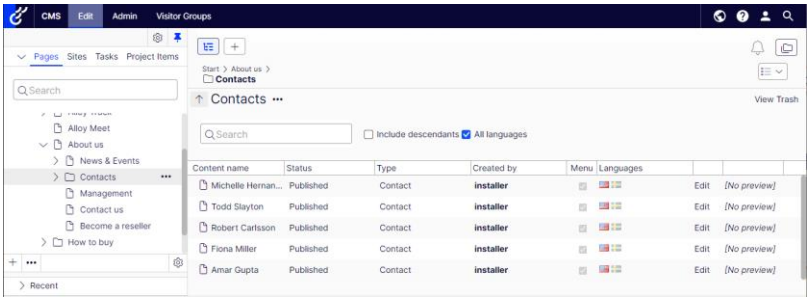
10. In the **Search** box, enter us and note the grid is filtered to only show pages that contain the word **us** in their name, as shown in the following screenshot:



11. In the **Assets** pane, select the **Alloy Plan** folder, in its context menu, select **Show grid**, and note the grid is populated with all the assets (both media and blocks) in that folder, as shown in the following screenshot:



12. In the **Pages** tree, select **Contacts**, and note that each contact page shows the columns **Contact name**, **Status**, **Type**, **Created by**, **Menu**, **Languages**, and actions to **Edit** and **Preview**, as shown in the following screenshot:



13. Click **Edit** for one of the contacts and note that a contact page has properties like **Phone** and **Email**.
14. Close the browser and shut down the web server.

Customizing grid view for a specific content type

Now we will customize the columns shown for a **ContactPage** to add columns like **Phone** and **Email**:

1. In **Startup.cs**, comment out the statement that enables grid view for all content, as shown in the following code:

```
services.AddGridView(options =>
{
   //options.IsViewEnabled = true;
});
```

2. In the **Business** folder, in the **UIDescriptors** folder, review the file named **ContainerPageUIDescriptor.cs**, and comment out the whole class, as shown in the following code:

```
using AlloyDemo.Models.Pages;
using EPiServer.Shell;

namespace AlloyDemo.Business.UIDescriptors
{
 /* please see the GridViews folder for replacement

 /// <summary>
 /// Describes how the UI should appear for <see cref="ContainerPage"/> content.
 /// </summary>
 [UIDescriptorRegistration]
 public class ContainerPageUIDescriptor : UIDescriptor<ContainerPage>
 {
  public ContainerPageUIDescriptor()
     : base(ContentTypeCssClassNames.Container)
  {
   DefaultView = CmsViewNames.AllPropertiesView;
  }
 }
 */
}
```

3. In the **Business** folder, add a new folder named **GridViews**.

4. In the **GridViews** folder, add a new class file named **ContainerPageGridViewUIDescriptor.cs**.
5. Modify the class to set grid view as the default view and to customize the columns for container pages to remove some standard columns and add **Phone** and **Email**, as shown in the following code:

```
using AlloyDemo.Models.Pages; // ContainerPage
using EPiServer.Labs.GridView; // ExtendedUIDescriptor<T>
using EPiServer.Labs.GridView.GridConfiguration; // GridSettings
using EPiServer.Shell; // [UIDescriptorRegistration]

namespace AlloyDemo.Business.GridViews
{
 [UIDescriptorRegistration]
 public class ContainerPageGridViewUIDescriptor
    : ExtendedUIDescriptor<ContainerPage>
 {
  public ContainerPageGridViewUIDescriptor()
     : base(ContentTypeCssClassNames.Container)
  {
   // show grid view by default
   DefaultView = SearchContentViewContentData.ViewKey;

   // customize the columns in the grid
   GridSettings = new GridSettings
   {

    Columns = new ColumnsListBuilder()
     .WithContentName()
     .WithRawPropertyValue(columnName: "Phone", propertyName: "Phone")
     .WithRawPropertyValue(columnName: "Email", propertyName: "Email")
     .WithContentStatus()
     .WithPublishDate()
     .WithEdit()
     .WithActionMenu()
     .Build()

    // other standard columns if you want them
    //.WithContentTypeName()
    //.WithCreatedBy()
    //.WithVisibleInMenu()
    //.WithCurrentLanguageBranch()
    //.WithPreviewUrl()
   };
  }
 }
}
```

6. In the **GridViews** folder, add a new class file named **ContainerPageGridView.cs**.
7. Modify the class to enable grid view for container pages, as shown in the following code:

```
using AlloyGridView.Models.Pages; // ContainerPage
using EPiServer.Labs.GridView; // SearchContentView<T>
using EPiServer.ServiceLocation; // [ServiceConfiguration]
using EPiServer.Shell; // ViewConfiguration

namespace AlloyDemo.Business.GridViews
```

```
{
    [ServiceConfiguration(typeof(ViewConfiguration))]
    public class ContainerPageGridView : SearchContentView<ContainerPage>
    {
    }
}
```

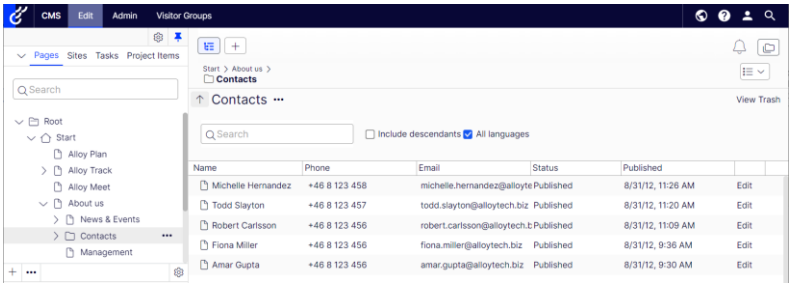
8. In the **Models** folder, in the **Pages** folder, in **ContactPage.cs**, import a namespace for working with grid view, as shown in the following code:

```
using EPiServer.Labs.GridView.ChildrenStore; // [IncludeInQueryResult]
```

9. Decorate the **Phone** and **Email** properties with an attribute to ensure it can be used as grid columns, as shown highlighted in the following code:

```
[Display(GroupName = Globals.GroupNames.Contact)]
[IncludeInQueryResult]
public virtual string Phone { get; set; }

[Display(GroupName = Globals.GroupNames.Contact)]
[EmailAddress]
[IncludeInQueryResult]
public virtual string Email { get; set; }
```

10. Start the **AlloyDemo** website project.
11. Log in as an administrator.
12. In **Edit** view, note product pages like **Alloy Plan** can only show **On-Page Editing** or **All Properties** views, the **How to buy** container page shows grid view with columns for **Phone** and **Email**, even though its children have **undefined** values for those columns, and the **Contacts** page shows a grid view with the **Phone** and **Email** values for its children, as shown in the following screenshot:



13. Close the browser and shut down the web server.

- Optimizely Gridview Customizations by Allan Thraen:
  https://www.codeart.dk/blog/2021/9/optimizely-gridview-customizations/

Module F – Working with Framework Components

Goal

The overall goal of the exercises in this module is to learn how to work with Framework components and content APIs. You will:

1. Export and import content.
2. Implement FAQs by programmatically creating pages using content APIs.
3. Validate content by listening for publishing events.
4. Process content by implementing a scheduled job.
5. Implement soft and hard deletes.
6. Decompile CMS assemblies to better understand how to work with our APIs.
7. Configure Content Delivery API integration for CMS and Search & Navigation.

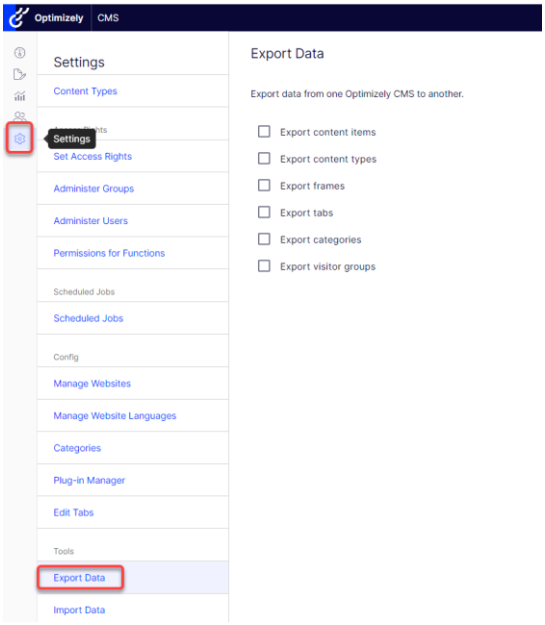
Exercise F1 – Exporting and importing content

In this exercise, you will export data from a CMS site, and then import it back in as if it were new content.

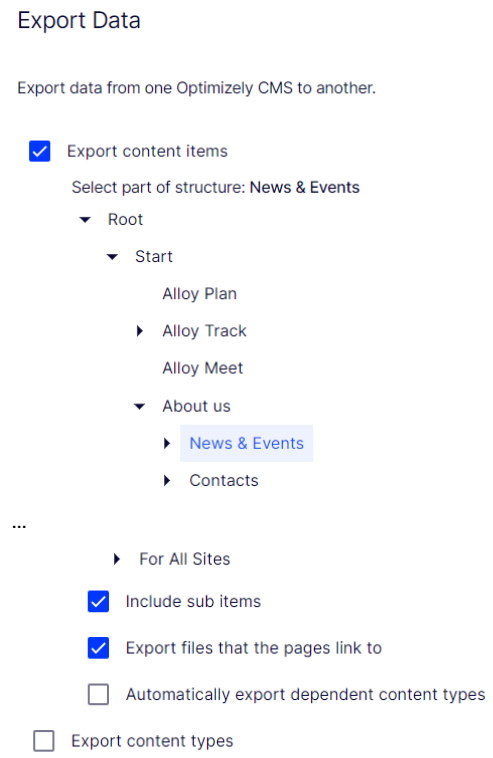
Prerequisites: complete Exercise A1.

Exporting news & events from the AlloyDemo website

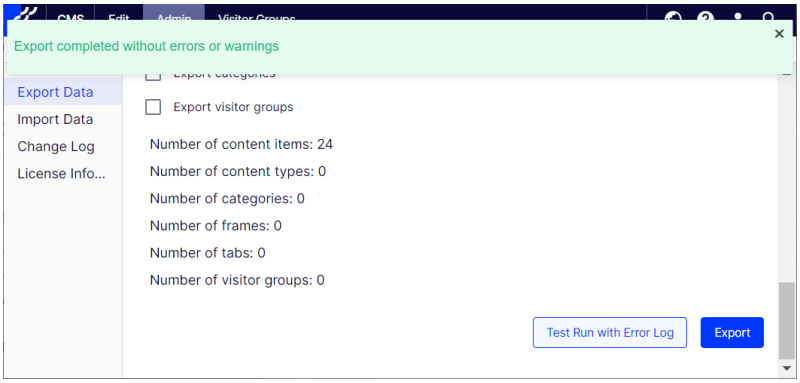
1. Open the solution with the **AlloyDemo** project.
   (This is the project that used the **Alloy MVC** project template, *not* the **Empty** template.)
2. Start the **AlloyDemo** website, and log in as **Admin**.
3. Navigate to **Settings** | **Export Data**, as shown in the following screenshot:



4. Select the **Export content items** check box, expand **Start** and **About us**, and then select **News & Events**, and note that by default the export will include sub items and any files (media assets) that the pages link to, as shown in the following screenshot:

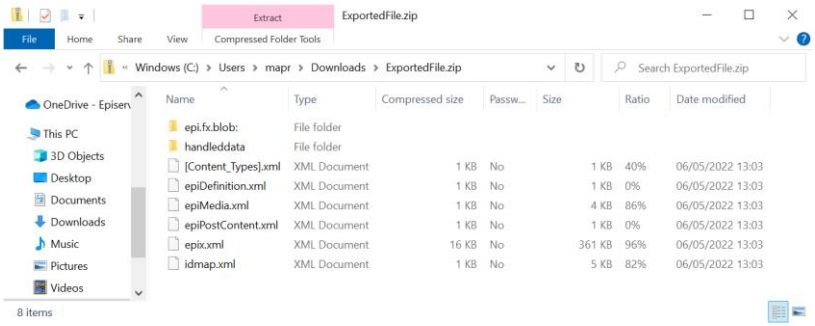


5. Click **Test Run with Error Log**, and note that you get no errors or warnings, as shown in the following screenshot:



6. Click **Export**, and note the file that was downloaded, as shown in the following screenshot:

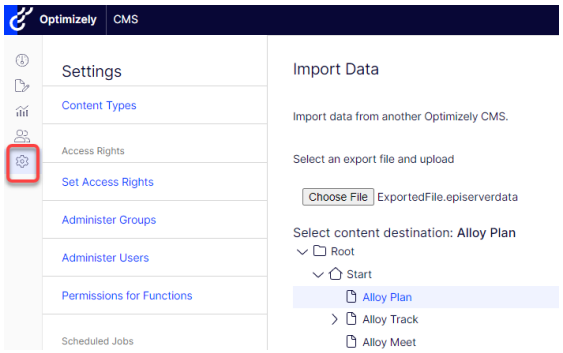
7. Start **File Explorer**, open your **Downloads** folder, and note that a file has been created named **ExportedFile.episerverdata**.
8. Copy the file and change its file extension to ZIP.
9. Open the ZIP file, and note the contents, as shown in the following screenshot:



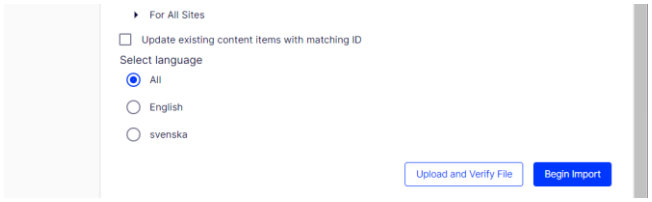
- The **epi.fx.blob:** folder contains the media assets like images used in the news & event articles exported. The **epix.xml** file contains the page and block content and their property values.

Importing news & events to AlloyDemo website

1. Start the **AlloyDemo** website, and log in as **Admin**.
2. Navigate to **Settings** | **Import Data**, choose the **ExportedFile.episerverdata** file, and select **Alloy Plan** as the content destination, as shown in the following screenshot:

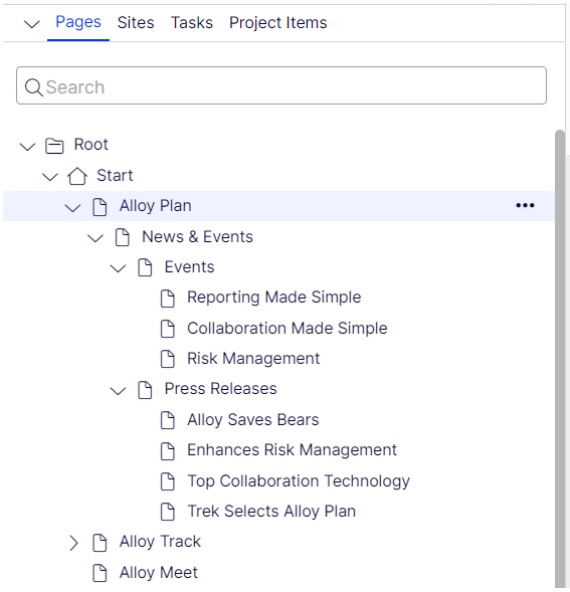


3. Clear the **Update existing content items with matching ID** check box, and then click **Begin Import**, as shown in the following screenshot:



- If you do not clear the **Update existing content items with matching ID** check box, then the pages will appear not to import, because it will just update existing items rather than add pages as new items!

4. After a few moments, the import should complete.
5. Navigate to **Edit**, and expand the **Pages** tree to see that **Alloy Plan** now has some new child pages, as shown in the following screenshot:



14. Close the browser and shut down the web server.

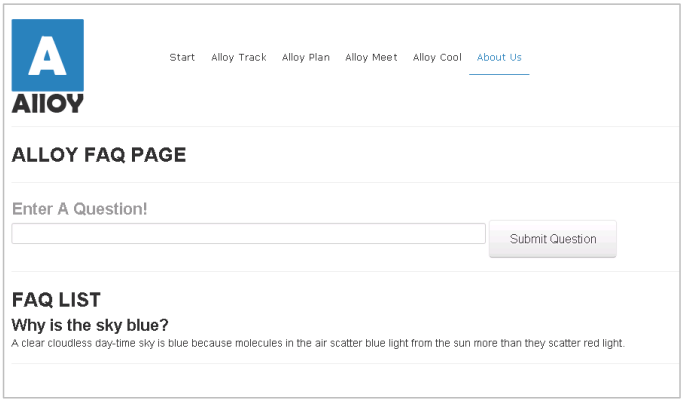
- For an interesting story around how CMS exports can fail, read the following article: https://getadigital.com/no/blogg/when-data-export-fails

Exercise F2 – Implementing FAQs with content APIs

In this exercise, you will implement FAQs by implementing data pages and automating their creation using the content repository API.

You will define two page types; one to be used as the visible **FAQListPage** containing both a form for visitors to enter a question and a listing of answered questions, and one named **FAQItemPage** that will act as a data page for a single question and its answer.

Here's an example of an FAQ page with one question answered:



Prerequisites: complete Exercise A1.

Create an FAQ item data page type and an FAQ list page type

The **FAQItemPage** will not have a template because it will just store data, but we want it to be created as a child of a **FAQListPage** that will have a page template that shows a list of its children.

1. Open the solution with the **AlloyDemo** project.
2. In **Models\Pages**, add a new class named **FAQItemPage.cs**.
3. Decorate the class with `[ContentType]`.
4. Change the **DisplayName** to **FAQ Item**.
5. Add a **Description** of "A data page for an FAQ item (cannot be created by editors)."
6. Add the **AvailableInEditMode** parameter and set to **false**.
7. Add the following properties with appropriate attributes:
    a. **Question:** `XhtmlString`
    b. **Answer**: `XhtmlString`

- **FAQItemPage** does not need an icon because content editors cannot see it in edit view. It does not need to inherit from `SitePageData` because it will not render as an HTML page with META elements.

Your code should look something like the following:

```
using EPiServer.Core; // PageData, XhtmlString
using EPiServer.DataAnnotations; // [ContentType]
using System.ComponentModel.DataAnnotations; // [Display]

namespace AlloyDemo.Models.Pages
{
```

```
    [ContentType(DisplayName = "FAQ Item",
        Description = "A data page for an FAQ item (cannot be created by editors).",
        AvailableInEditMode = false)]
    public class FAQItemPage : PageData
    {
        [Display(Name = "Question", Order = 10)]
        public virtual XhtmlString Question { get; set; }

        [Display(Name = "Answer", Order = 20)]
        public virtual XhtmlString Answer { get; set; }
    }
}
```

8. In **AlloyDemo**, in **Models\Pages**, add a new class named **FAQListPage.cs**.
9. Decorate the class with `[ContentType]`.
10. Change the **DisplayName** to **FAQ List**.
11. Group the page type under **Specialized** by using a constant in **Global.GroupNames**.
12. Add a **Description** of "Use this page for a list of FAQs entered by visitors, answered by editors."
13. Apply the **SiteImageUrl** attribute to the class.
14. Apply an attribute to restrict this page types children to only be **FAQItemPage** instances.
15. Inherit from **SitePageData**.
16. Delete the **MainBody** property.
17. Add a property named **FAQItems** that can contain the child FAQ item pages and apply the **Ignore** attribute to it.

- Good practice would be to define a view model to hold this property but marking a content type property with **[Ignore]** means the property will not be registered with the CMS so it can be used for a similar purpose as a view model. Just beware of object caching!

Your code should look something like the following:

```
using EPiServer.DataAnnotations;
using System.Collections.Generic;

namespace AlloyDemo.Models.Pages
{
    [ContentType(DisplayName = "FAQ List",
        GroupName = Globals.GroupNames.Specialized,
        Description = "Use this page for a list of FAQs entered by visitors, answered by editors.")]
    [SiteImageUrl]
    [AvailableContentTypes(Include = new[] { typeof(FAQItemPage) },
        IncludeOn = new[] { typeof(StartPage) })]
    public class FAQListPage : SitePageData
    {
        // having an ignored property avoids needing a view model
        // this property will not be stored in CMS so it does not
        // need to be virtual
        [Ignore]
        public IEnumerable<FAQItemPage> FAQItems { get; set; }
    }
}
```

Creating an FAQ list page template

1. In **AlloyDemo**, in **Controllers**, add a new class named **FAQListPageController.cs**.
2. Modify the class to derive from **PageControllerBase<T>**.
3. Modify the **Index** action method to use a view model and set the **FAQItems** property using the content loader service.
4. Add a **CreateFAQItem** method, with two parameters: **currentPage** and **question**.
5. Use the content repository service to add a new FAQ item page under the current page, setting its **Question** property to the **question** parameter, and giving it a **Name**.
6. Save and check in the new FAQ item page if the current user has **Read** access rights.

Your controller should look something like the following code:

```
using AlloyDemo.Models.Pages;
using AlloyDemo.Models.ViewModels;
using EPiServer;
using EPiServer.Core;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace AlloyDemo.Controllers
{
    public class FAQListPageController : PageControllerBase<FAQListPage>
    {
        protected readonly IContentRepository repo;

        public FAQListPageController(IContentRepository repo)
        {
            this.repo = repo;
        }

        public IActionResult Index(FAQListPage currentPage)
        {
            PageViewModel<FAQListPage> viewmodel =
                PageViewModel.Create(currentPage);
            IEnumerable<FAQItemPage> faqs = repo.GetChildren<FAQItemPage>(
                currentPage.ContentLink);
            viewmodel.CurrentPage.FAQItems = faqs;
            return View(viewmodel);
        }

        public IActionResult CreateFAQItem(
            FAQListPage currentPage, string question)
        {
            FAQItemPage faqItem =
                repo.GetDefault<FAQItemPage>(currentPage.ContentLink);

            // if someone is logged in then CreatedBy and ChangedBy will be set,
            // otherwise they will be empty string which is shown as "installer"
            if (string.IsNullOrEmpty(faqItem.CreatedBy))
                faqItem.CreatedBy = "Anonymous";
            if (string.IsNullOrEmpty(faqItem.ChangedBy))
                faqItem.ChangedBy = "Anonymous";

            faqItem.Question = new XhtmlString(question);
```

```
            faqItem.Name = "Q. " + question;
            repo.Save(faqItem,
                EPiServer.DataAccess.SaveAction.CheckOut,
                EPiServer.Security.AccessLevel.Read);

            return RedirectToContent(currentPage.ContentLink, "Index");
        }
    }
}
```

Commented [BG3]: Had to change this line from 'RedirectToAction('Index'), due to changes in .NET 6 routing the old version would generate null reference errors.

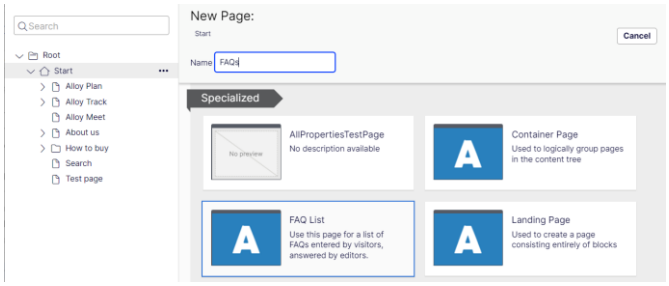
7. In **AlloyDemo**, in **Views**, add a new folder named **FAQListPage**.
8. In **FAQListPage**, add a new empty Razor view named **Index.cshtml**.
9. Modify the view, as shown in the following markup, and note the following:
    a. The question form submits back to the **CreateFAQItem** action method.
    b. Content editors see a warning message.
    c. Each FAQ item outputs its: question and when it was asked, and the answer and who and when it was answered.

```
@inject IContextModeResolver modeResolver
@model AlloyDemo.Models.ViewModels.PageViewModel<FAQListPage>
<h2>Alloy FAQs</h2>
<div class="navbar">
    <h3>Enter your question</h3>
    @using (Html. BeginContentForm(actionName: "CreateFAQItem"))
    {
        <input type="text" tabindex="1" name="question"
               placeholder="Enter your question" />
        <input type="submit" tabindex="2" class="btn" value="Submit" />
    }
</div>
@if (modeResolver.CurrentMode == ContextMode.Edit)
{
    <div class="alert alert-info">Questions do not appear until they have been answered by a content editor
and published.</div>
}
<div class="nav-stacked">
    <h2>Questions with answers</h2>
    @foreach (var faqItem in Model.CurrentPage.FAQItems)
    {
    <div class="alert alert-success">
        <p><small class="label label-inverse">Asked at @faqItem.Created.ToShortTimeString() on
@faqItem.Created.ToShortDateString()</small></p>
        @Html.DisplayFor(m => faqItem.Question)
        <p><small class="label label-inverse">Answered by @faqItem.ChangedBy at
@faqItem.StartPublish.Value.ToShortTimeString() on
@faqItem.StartPublish.Value.ToShortDateString()</small></p>
        @Html.DisplayFor(m => faqItem.Answer)
    </div>
    }
</div>
```

Commented [BG4]: Had to change this line from 'Html.BeginForm(actionName: "CreateFAQItem", controllerName: null)', the original doesn't work in .NET 6 and the action never gets called. Might want to verify this with different revisions of CMS, the version I was working with was 12.22.0

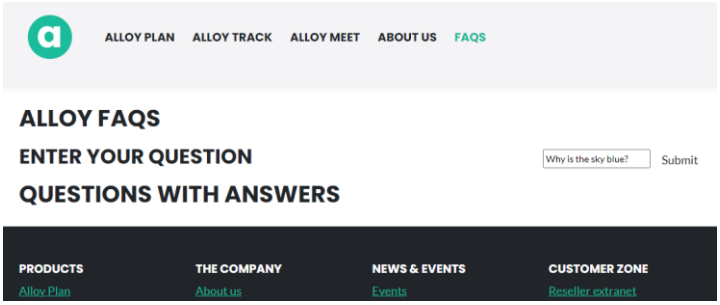
Test the website FAQ functionality

1. Start the **AlloyDemo** website, and log in as **Admin**.
2. Create an FAQ list page named **FAQs** under the **Start** page, as shown in the screenshot:

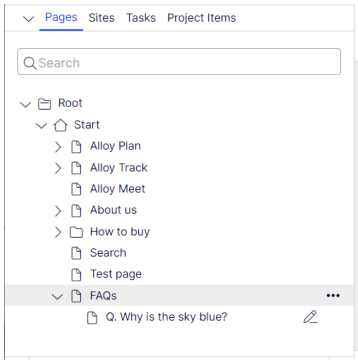


3. **Publish** the page.
4. Switch to **Live** view and log out to experience the website as an anonymous visitor, as shown in the following screenshot:

5. Navigate to the **FAQs** page.
6. Enter and submit a question, for example, “Why is the sky blue?”, and note that the FAQ does NOT appear, as shown in the following screenshot:



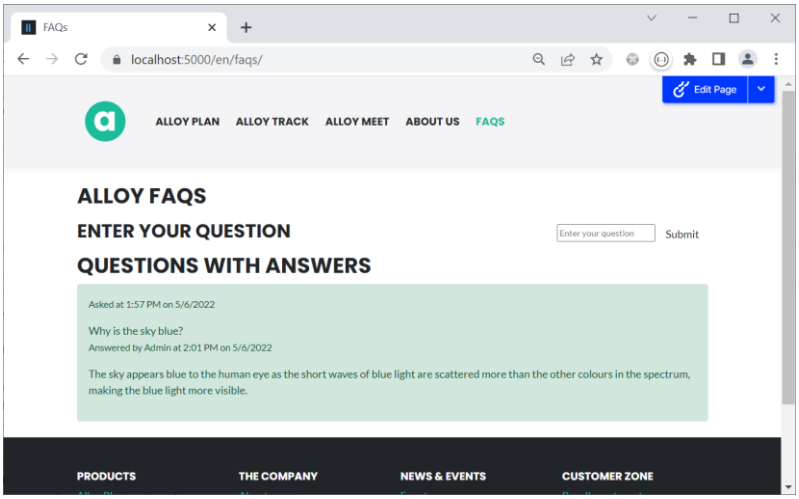
7. Log in as **Admin**, view the **Pages** tree, and note the **FAQs** page has a child with a pencil icon to indicate that it has a draft that has not been published yet, as shown in the screenshot:



8. Edit the question, answer it, and publish it.

- The sky appears blue to the human eye as the short waves of blue light are scattered more than the other colours in the spectrum, making the blue light more visible.

9. Switch to Live view, and note the question and its answer now appear, as shown in the following screenshot:



10. Close the browser and shut down the web server.

Exercise F3 – Listening for events and customizing services with initialization modules

Commented [BG5]: Look into replacing this using the more current .NET and CMS 12 techniques.

In this exercise, you will create initialization modules that (1) prevents creating a page if the parent page already has more than a maximum number of children, and (2) removes the service that suggests most recent content types.

Prerequisites: complete Exercise A1.

Creating an initialization module to listen for events

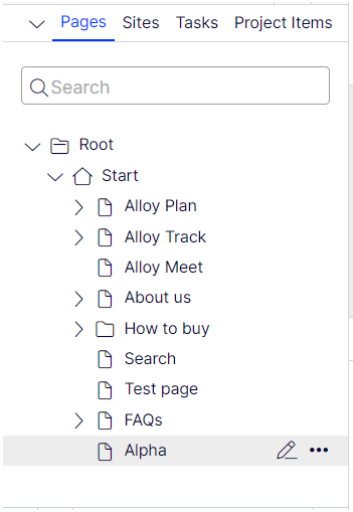
1. In **AlloyDemo**, in **Business\Initialization**, add a new class file named **PreventMoreThanMaxChildrenInitializationModule.cs**.
2. Modify the statements, as shown in the following code:

```
using AlloyDemo.Models.Pages; // SitePageData
using EPiServer; // IContentLoader
using EPiServer.Core; // IContentEvents, IContent
using EPiServer.Framework; // [InitializableModule], [ModuleDependency]
using EPiServer.Framework.Initialization; // InitializationEngine
using System.Linq; // Count()

namespace AlloyDemo.Business.Initialization
{
    [InitializableModule]
    [ModuleDependency(typeof(EPiServer.Web.InitializationModule))]
    public class PreventMoreThanMaxChildrenInitializationModule :
        IInitializableModule
    {
        private bool initialized = false;
        private IContentEvents events;
        private IContentLoader loader;
        private const int maxChildren = 8;

        public void Initialize(InitializationEngine context)
        {
            if (!initialized)
            {
                loader = context.Locate.ContentLoader();
                events = context.Locate.ContentEvents();
                events.CreatingContent += Events_CreatingContent;
                initialized = true;
            }
        }

        private void Events_CreatingContent(object sender, ContentEventArgs e)
        {
            var sitepage = e.Content as SitePageData;
            if (sitepage != null)
            {
                var children = loader.GetChildren<IContent>(sitepage.ParentLink);
                if (children.Count() >= maxChildren)
                {
                    e.CancelAction = true;
                    e.CancelReason =
```

```
            $"Cannot create a new page if the parent has {maxChildren} or more children.";
                }
            }
        }

        public void Uninitialize(InitializationEngine context)
        {
            if (initialized)
            {
                events.CreatingContent -= Events_CreatingContent;
            }
        }
    }
}
```

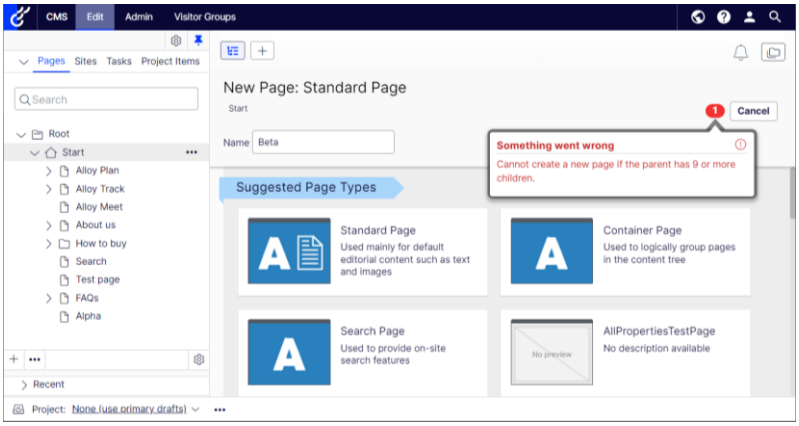
- Change the maximum allowed to one more than the current number of children under the Start page.

3. Start the site, and log in as **Admin**.
4. Navigate to **CMS** | **Edit**.
5. Add a **Standard** page underneath the **Start** page named **Alpha**. You will be able to create the page, as shown in the following screenshot:



6. **Publish** the Alpha page.

7. Add a **Standard** page underneath the **Start** page named **Beta**. This time, you will not be able to create the page, because Start now has nine children, as shown in the following screenshot:



8. Click **Cancel**.
9. Close the browser and shut down the web server.

- Limiting the number of children to eight will cause you annoyance later, so change the limit in the initialization module to something like 100, as shown in the following code:

```
private const int maxChildren = 100;
```

Creating an initialization module to remove a service

Commented [BG6]: Look into doing this using more current .NET and CMS 12 techniques.

1. In the previous screenshot, note the **Suggested Page Types** group. This group is automatically created based on the *most recently used* page types.
2. In **Business\Initialization**, add a new class file named **RemoveSuggestedPageTypesInitializationModule.cs**.
3. Modify the statements, as shown in the following code:

- Note the class implements **IConfigurableModule** (that itself implements **IInitializationModule**), and that the method we implement to remove the service is named **ConfigureContainer**. Both the methods **Initialize** and **Uninitialize** must exist, but in this scenario, they do nothing.

```
using EPiServer.Cms.Shell.UI.Rest; // IContentTypeAdvisor
using EPiServer.Framework; // [InitializableModule], [ModuleDependency]
using EPiServer.Framework.Initialization; // InitializationEngine
using EPiServer.ServiceLocation; // IConfigurableModule, ServiceConfigurationContext
using System.Linq; // FirstOrDefault()

namespace AlloyDemo.Business.Initialization
{
    [InitializableModule]
    [ModuleDependency(typeof(EPiServer.Web.InitializationModule))]
    public class RemoveSuggestedPageTypesInitializationModule :
        IConfigurableModule
    {
        public void ConfigureContainer(ServiceConfigurationContext context)
        {
```

```
            var descriptor = context.Services.FirstOrDefault(
                d => d.ServiceType == typeof(IContentTypeAdvisor));

            if (descriptor != null)
            {
                context.Services.Remove(descriptor);
            }
        }

        public void Initialize(InitializationEngine context) { }

        public void Uninitialize(InitializationEngine context) { }
    }
}
```

4. Start the site, and log in as **Admin**.
5. Navigate to **CMS** | **Edit**.
6. Add a **Standard** page underneath the **Start** page, and note the **Suggested Page Types** group has gone.

Exercise F4 – Implementing scheduled jobs

In this exercise, you will implement a scheduled job to modify page names, titles, and descriptions that contain bad words. This exercise covers:

- Implementing a scheduled job.
- Using IContentRepository to programmatically modify content.
- Using IPageCriteriaQueryService to programmatically find content,

Prerequisites: complete Exercise A1.

Create a scheduled job

1. Open the solution that includes the **AlloyDemo** project.
2. In the **AlloyDemo** project, in **Startup.cs**. in the `ConfigureServices` method, comment out the statement that disables scheduled jobs, as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    if (_webHostingEnvironment.IsDevelopment())
    {
        AppDomain.CurrentDomain.SetData("DataDirectory", Path.Combine(
            _webHostingEnvironment.ContentRootPath, "App_Data"));

        // services.Configure<SchedulerOptions>(
        //     options => options.Enabled = false);
    }
```

3. In **Business**, add a folder named **ScheduledJobs**, and add a new class file named **BadWordScannerScheduledJob.cs**.
4. Modify its contents, as shown in the following code:

```
using AlloyDemo.Models.Pages; // SitePageData
using EPiServer; // IContentRepository, PropertyCriteriaCollection, PropertyCriteria
using EPiServer.Core; // IPageCriteriaQueryService, PropertyDataType, PageDataCollection,
ContentReference, PageReference
using EPiServer.PlugIn; // [ScheduledPlugIn]
using EPiServer.Scheduler; // ScheduledJobBase
using EPiServer.ServiceLocation; // ServiceLocator

namespace AlloyDemo.Business.ScheduledJobs
{
    [ScheduledPlugIn(DisplayName = "Bad Word Scanner",
        Description = "Scan for bad words in pages and censor them.")]
    public class BadWordScannerScheduledJob : ScheduledJobBase
    {
        private bool _stopSignaled;

        // you could load this from a file or service
        private string[] badWords = new[] { "frak" };

        public BadWordScannerScheduledJob()
        {
            IsStoppable = true;
        }
```

Commented [BG7]: My demo site didn't have this disabled, not sure if it was a step from Exercise A1 that I skipped, need to check.

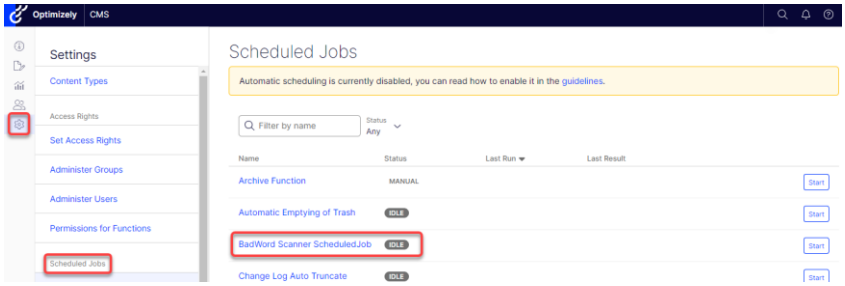
Commented [BG8]: Could add instructions to use 'dotnet new epi-cms-job -n BadWordScannerScheduledJob -p:na AlloyDemo.Business.ScheduledJobs'

```
public override void Stop()
{
    _stopSignaled = true;
}

public override string Execute()
{
    OnStatusChanged(string.Format(
        "Starting execution of {0}", this.GetType()));

    var repo = ServiceLocator.Current.GetInstance<IContentRepository>();

    var finder = ServiceLocator.Current
        .GetInstance<IPageCriteriaQueryService>();

    int pageCount = 0;

    foreach (string word in badWords)
    {
        var criteria = new PropertyCriteriaCollection();

        criteria.Add(new PropertyCriteria
        {
            Type = PropertyDataType.LongString,
            Name = "PageName",
            Condition = EPiServer.Filters.CompareCondition.Contained,
            Value = word
        });

        criteria.Add(new PropertyCriteria
        {
            Type = PropertyDataType.LongString,
            Name = "MetaTitle",
            Condition = EPiServer.Filters.CompareCondition.Contained,
            Value = word
        });

        criteria.Add(new PropertyCriteria
        {
            Type = PropertyDataType.LongString,
            Name = "MetaDescription",
            Condition = EPiServer.Filters.CompareCondition.Contained,
            Value = word
        });

        PageDataCollection results = finder.FindPagesWithCriteria(
            ContentReference.RootPage as PageReference, criteria);

        foreach (SitePageData page in results)
        {
            var clone = page.CreateWritableClone() as SitePageData;

            clone.Name = page.Name.Replace(word, "[censored]");
            clone.MetaTitle =
                page.MetaTitle?.Replace(word, "[censored]");
```

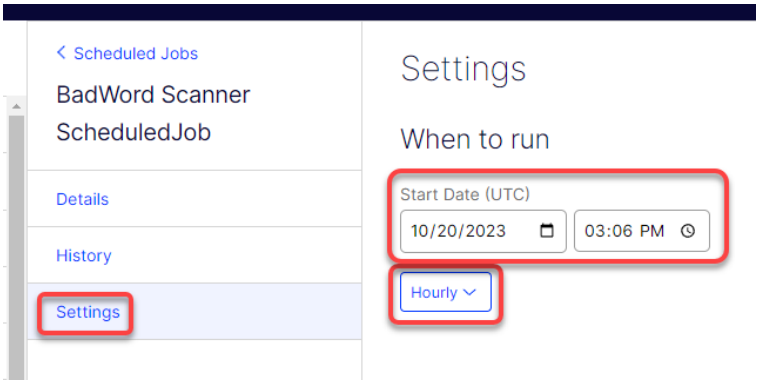
```
                clone.MetaDescription =
                    page.MetaDescription?.Replace(word, "[censored]");

                repo.Save(clone,
                    EPiServer.DataAccess.SaveAction.CheckIn,
                    EPiServer.Security.AccessLevel.NoAccess);

                pageCount++;
            }

            //For long running jobs periodically check if stop is signaled and if so stop execution
            if (_stopSignaled)
            {
                return "Stop of job was called";
            }
        }

        return $"{pageCount} pages containing one of the following bad words: '{string.Join("' or '",
badWords)}' have been censored.";
    }
  }
}
```

Testing the scheduled job

1. Start the **AlloyDemo** site, and log in as **Admin**.
2. Edit **Start**, add **frak** into its **Name**, and publish the change.
3. Edit **About us**, add **frak** into its **Title** (MetaTitle), and publish the change.
4. Edit **Alloy Track**, add **frak** into its **Page description** (MetaDescription), and publish the change.
5. Navigate to **Settings** | **Scheduled Jobs** | **Bad Word Scanner**, as shown in the following screenshot:



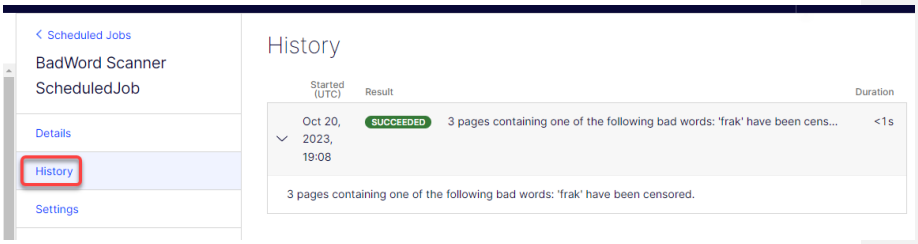
6. Set the job to run every hour, set it to be active, and change the next schedule date to two minutes in the future, as shown in the following screenshot:



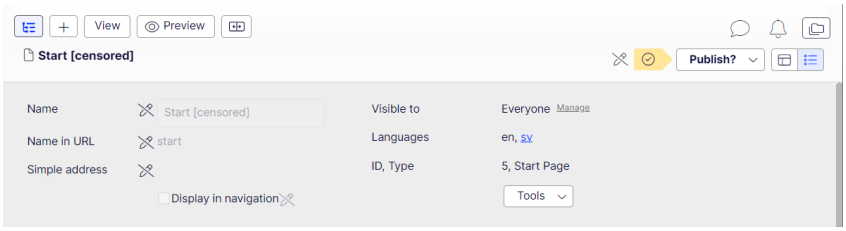
7. Click the **Save Scheduled Job** button.
8. Log out, view the website as an anonymous visitor, and wait for the time you set to pass.
9. At the command line or terminal, note the success message, as shown in the following output:

```
info: EPiServer.DataAbstraction.ScheduledJob[0]
      Successfully executed the scheduled job 'Bad Word Scanner' with jobId
='4ccf7a86-dc63-43ca-a5c5-8bc9e735967a'
```

10. Navigate to **Settings** | **Scheduled Jobs** | **Bad Word Scanner**.
11. Click **History**, and note the job ran succesfully and censored three pages, as shown in the following screenshot:



12. Navigate to **Edit**, and note the Start page has been censored and marked as being ready to publish, as shown in the following screenshot:



13. **Publish** the censored change.
14. Check the **About us** and **Alloy Track** pages are similarly censored and ready to publish, and then publish their changes.
15. Close the browser and shut down the web server.

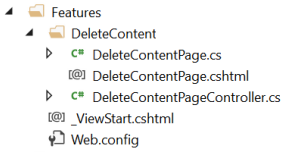
Exercise F5 – Implementing soft and hard deletes

In this exercise, you will create a page type with template to allow anonymous visitors to enter a content reference of an item of content to delete.

Prerequisites: complete Exercise A1.

Adding the delete content feature

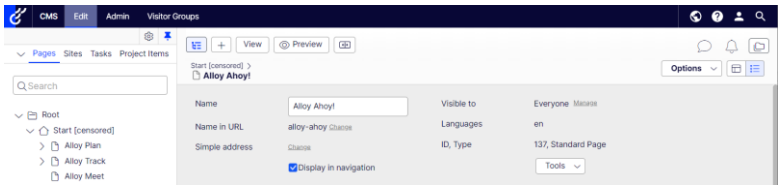
1. Drag and drop the **cmsdevfun-exercisefiles\Module F\F5\Features** folder into the **AlloyDemo** project.
2. Expand the **Features** folder and review the files included, as shown in the following screenshot:



   a. **DeleteContentPage.cs**: a page type class to enable the feature.
   b. **DeleteContentPage.cshtml**: a Razor file for the user interface of the feature.
   c. **DeleteContentPageController.cs**: a controller that performs the work of the feature.
3. In **DeleteContentPageController.cs**, note the following:
   a. The **content repository service** set using constructor parameter injection. It will be used to either delete (if hard delete is "on") or move the content reference to the wastebasket.
   b. The **Delete** action method that responds to HTTP POSTs, with three parameters: **currentPage**, **contentReference**, and **hardDelete**. If hardDelete is "on" then it uses the content repository's `Delete()` method to permanently delete the content identified with the contentReference, else it uses the `Move()` method to move the content to the Wastebasket. It sets a message in **ViewData** to tell the visitor what happened.
4. Open **DeleteContentPage.cshtml** and note the following:
   a. If a message is passed using **ViewData**, it is shown to the visitor.
   b. A form that allows a visitor to enter a content reference, select a check box for hard delete, and click a **Delete** button.

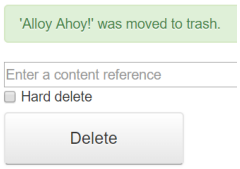
Test the website delete content functionality

1. Start the **AlloyDemo** website, and log in as **Admin**.
2. Create and publish a **Delete Content** page named **Deleter** under the **Start** page.
3. Create and publish a **Standard** page named **Alloy Ahoy!** under the **Start** page.
4. Make a note of the ID for **Alloy Ahoy!**, for example 137, as shown in the following screenshot:

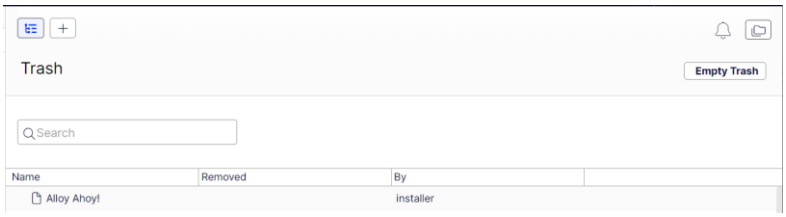


5. Switch to **Live** view and log out so that you are anonymous.
6. Navigate to the **Deleter** page.

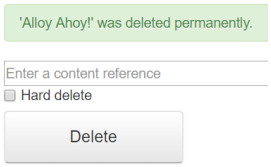
7. Enter the ID of the Alloy Ahoy! page, click **Delete**, and note the message, as shown in the following screenshot:



8. Log in as **Admin**, and view the **Trash**, as shown in the following screenshot:



9. **Restore** the Alloy Ahoy! page.
10. Switch to **Live** view and log out so that you are anonymous.
11. Navigate to the **Deleter** page.
12. Enter the ID of the Alloy Ahoy! page, select the **Hard delete** check box, click **Delete**, and note the message, as shown in the following screenshot:



13. Log in as Admin and confirm that the page is not in the **Trash**, or the **Pages** tree, and has been permanently deleted.
14. Close the browser and shut down the web server.

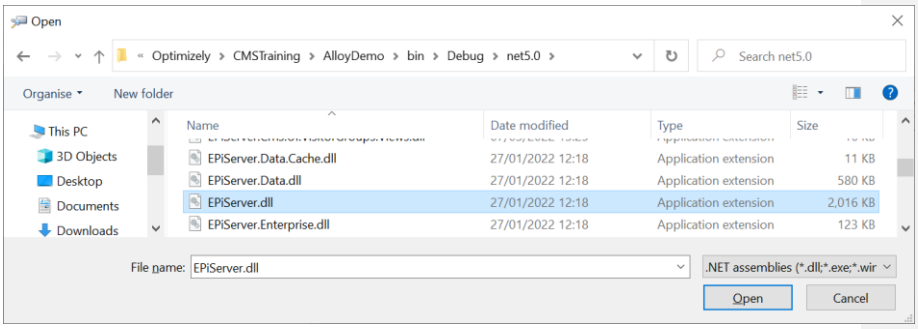
Exercise F6 – Learning from CMS assemblies

In this exercise, you will use ILSpy to decompile and view CMS assembly code to learn from it.

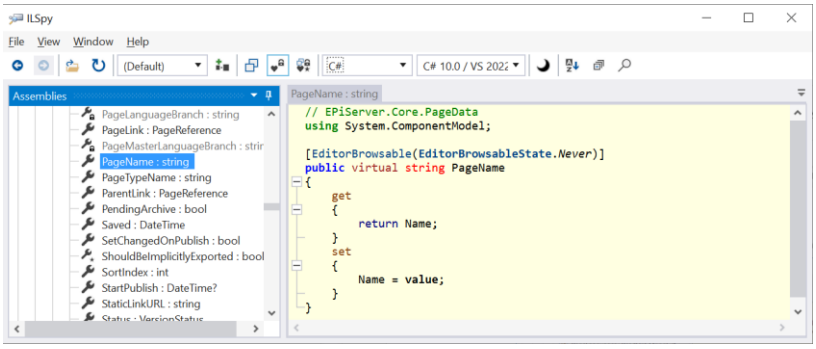
Prerequisites: complete Exercise A1.

Decompiling CMS assemblies to understand our APIs

1. Download and install **ILSpy** from: https://github.com/icsharpcode/ILSpy/releases/latest
2. Start **ILSpy**, choose **File** | **Open**, browse to **AlloyDemo** site's **bin\Debug\net6.0** folder, select **EPiServer.dll**, and click **Open**, as shown in the following screenshot:

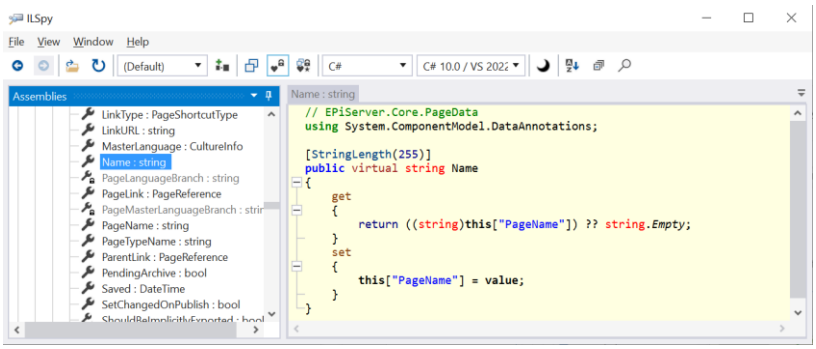


3. Expand the **EPiServer (12.x.x.x, .NETCoreApp, v6.0)** assembly.
4. Expand the **EPiServer.Core** namespace.
5. Expand the **PageData** class.
6. Select **PageName**, and note that property is decompiled, as shown in the following screenshot:



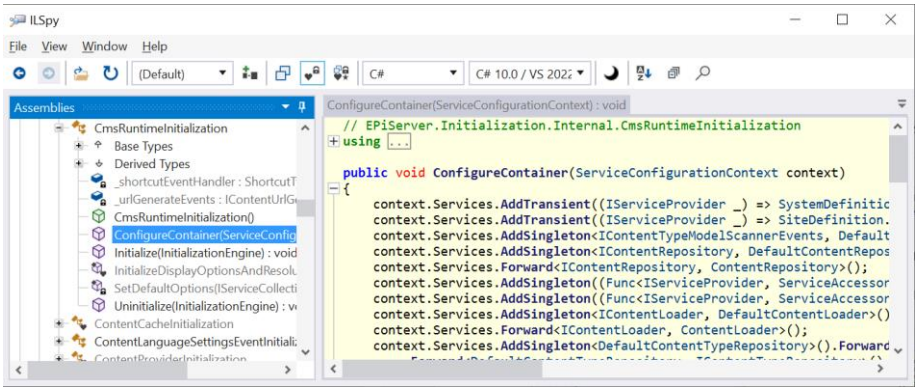
- `[EditorBrowsable(EditorBrowsableState.Never)]` hides the `PageName` property in IntelliSense because it is legacy and should be avoided in favor of `Name`.

7. In the decompiled source code, click **this.Name**, and note that property is decompiled, and that internally the property still uses `PageName`, as shown in the following screenshot:



Reviewing CMS initialization

1. In the left list, collapse **EPiServer.Core**.
2. Scroll down the tree view and expand **EPiServer.Initialization.Internal**.
3. Expand the **CmsRuntimeInitialization** class.
4. Select the method named **ConfigureContainer(ServiceConfigurationContext) : void**.
5. Note some of the things it does, like registering the **IContentLoader** dependency service to be a singleton instance of **DefaultContentLoader**, as shown in the following screenshot:



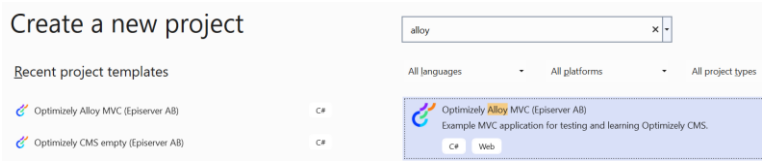
- You have now learned how to use tools like ILSpy to see how our own developers implement functionality.

Exercise F7 – Adding Headless with Content Delivery API

Prerequisites: Install Visual Studio 2022 and Optimizely Templates.

Creating an Alloy website project

1. In Visual Studio 2022, create a new project.
2. Search for and select **Optimizely Alloy MVC (Episerver AB)**, as shown in the following screenshot:



3. For **Project name**, enter `Alloy`, for **Solution name**, enter `Headless`, and select your preferred location.
4. Click **Next** and then **Create**.
5. Navigate to **Debug** | **Start Without Debugging**.
6. Wait for the project to build and for the browser to connect to the website.
7. You will be automatically redirected to `https://localhost:5000/Util/Register` where you can register an administrator account.
    a. For **Username**, enter `Admin`.
    b. For **Email**, enter `admin@alloy.com`.
    c. For **Password**, enter `Pa$$w0rd`.
8. When the Alloy start page appears, close the browser, and shut down the Kestrel web server.

Adding Search & Navigation and Content Delivery API

1. In **Alloy.csproj**, update the package references and add new package references, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>disable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <Using Include="EPiServer" />
    <Using Include="EPiServer.Core" />
    <Using Include="EPiServer.DataAbstraction" />
    <Using Include="EPiServer.DataAnnotations" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="EPiServer.CMS" Version="12.7.0" />
    <PackageReference Include="EPiServer.Find.Cms" Version="14.0.4" />
    <PackageReference Include="EPiServer.ContentDeliveryApi.Cms"
                      Version="3.3.0" />
    <PackageReference Include="EPiServer.ContentDeliveryApi.Search"
```

Commented [BG9]: You get version conflicts with the later CMS version used by the dotnet templates. This exercise should either be removed or use the Package manager UI to update the CMS and install the latest versions of the packages listed. Once that's done the rest works pretty well although there could be a bit more instruction on how to use the http file in Code to run the requests.

```
                Version="3.3.0" />
    <PackageReference Include="EPiServer.Framework" Version="12.5.0" />
    <PackageReference Include="EPiServer.Hosting" Version="12.5.0" />
    <PackageReference Include="Wangkanai.Detection" Version="5.2.1" />
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resources\Translations\**\*" />
  </ItemGroup>
</Project>
```

2. In `appsettings.Development.json`, add configuration for your Search & Navigation index, as shown highlighted in the following configuration:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "EPiServer": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "ConnectionStrings": {
    "EPiServerDB": "Data
Source=(LocalDb)\\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\\AlloyDemo.mdf;Initial
Catalog=AlloyDemo;Integrated Security=True;Connect Timeout=30"
  },
  "EPiServer": {
    "Cms": {
      "MappedRoles": {
        "Items": {
          "CmsEditors": {
            "MappedRoles": [ "WebEditors", "WebAdmins" ]
          },
          "CmsAdmins": {
            "MappedRoles": [ "WebAdmins" ]
          }
        }
      }
    },
    "Find": {
      "ServiceUrl": "https://es-eu-api01.episerver.net/PlpMSGvtyIh92X...xGRv",
      "DefaultIndex": "episervertraining_..."
    }
  }
}
```

- Your `ServiceUrl` and `DefaultIndex` will be different. You can register for a free 30-day index at the following link: `https://find.episerver.com/`

3. In `Startup.cs`, in the `ConfigureServices` method, add statements to configure Search & Navigation and Content Delivery API, including enabling friendly URLs, limiting search results, and enabling site definitions, as shown highlighted in the following code:

   services

```
    .AddCmsAspNetIdentity<ApplicationUser>()
    .AddCms()
    .AddFind() // enable Search & Navigation
    .AddAlloy()
    .AddAdminUserRegistration()
    .AddEmbeddedLocalization<Startup>();

services.AddContentDeliveryApi(options =>
{ // enable Content Delivery API including sites
  options.SiteDefinitionApiEnabled = true;
})
.WithFriendlyUrl(); // enable friendly URLs

services.AddContentSearchApi(options =>
{ // enable Content Search API and limit results
  options.MaximumSearchResults = 10;
});
```

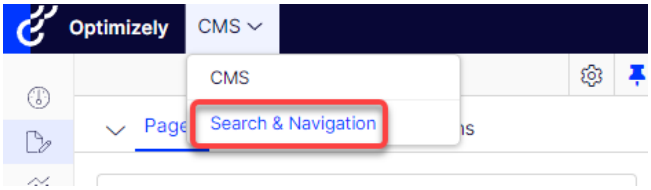
4. In `Startup.cs`, in the `Configure` method, before the call to `UseEndpoints`, add a statement to enable CORS, as shown highlighted in the following code:

```
// required for Content Search API
app.UseCors();

app.UseEndpoints(endpoints =>
{
  endpoints.MapContent();
});
```

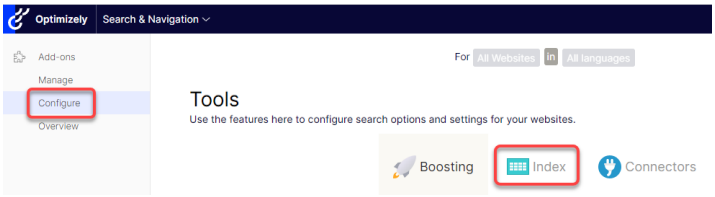
5. Start the **Alloy** project.
6. Log in as **Admin**.
7. Click **Edit Page**.

- It is necessary to reindex the website to ensure all the documents in Search & Navigation have the extra fields included needed by Content Delivery API.

8. From the CMS dropdown menu select **Search & Navigation**.

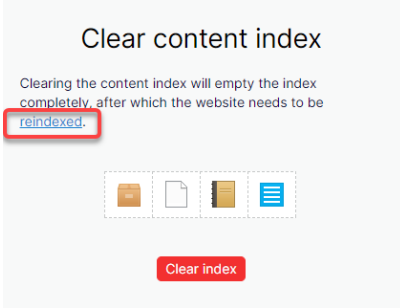


9. Navigate to **Search & Navigation** | **Configure** | **Index**, as shown in the following screenshot:



10. Click the **Clear Index** button and in the warning dialog box click **Clear content index**.

11. Click the **reindexed** link to navigate to the scheduled job page, as shown in the following screenshot:



12. From the Search & Navigation Content Indexing Job page, click **Start**.
13. Wait for the job to successfully finish.
14. Start **Visual Studio Code**.
15. In the **Headless** folder, create a folder named **RESTClient**, and open it.
16. In the **RESTClient** folder, create a file named `get-alloy-content.http`.
17. Add the following HTTP requests:

```
### Get the sites
GET https://localhost:5000/api/episerver/v3.0/site/


### Get the Start page
GET https://localhost:5000/api/episerver/v3.0/content/5


### Get the Start page using its ContentGuid
GET https://localhost:5000/api/episerver/v3.0/content/7b08c7d5-7585-47e5-add7-
5822da68c3ce


### Get the Start page in Swedish
GET https://localhost:5000/api/episerver/v3.0/content/5
Accept-Language: sv


### Get the About us page whose ContentLink.Id is 10
GET https://localhost:5000/api/episerver/v3.0/content/10


### Get the About us page using its friendly URL
GET https://localhost:5000/en/about-us/
Accept: application/json


### Get the Start page using its friendly URL in English
GET https://localhost:5000/en/
```

```
Accept: application/json

### Get the Start page using its friendly URL in Swedish
GET https://localhost:5000/sv/
Accept: application/json

### Get the children of the About us page
GET https://localhost:5000/api/episerver/v3.0/content/10/children

### Get the ancestors of the About us page
GET https://localhost:5000/api/episerver/v3.0/content/10/ancestors

### Get the Start and About us pages
GET https://localhost:5000/api/episerver/v3.0/content/?references=5,10

### Search for everything
GET https://localhost:5000/api/episerver/v3.0/search/content

### Search for "white"
GET https://localhost:5000/api/episerver/v3.0/search/content?query=white
```

18. Test all the requests to make sure they return the expected results.
19. Close the folder, quit Visual Studio Code, and shut down the Kestrel web server.

Module G – Optimizing, Securing, and Deploying

Goal

The overall goal of the exercises in this module is to learn how to prepare a CMS website before deployment. You will:

1. Control the caching of responses in CDN and browser.
2. Implement logging.
3. Secure the AlloyDemo website.

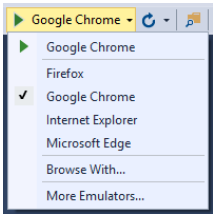
Exercise G1 – Controlling the caching of responses

In this exercise, you will set cache-control headers and confirm that they were sent in the HTTP response to be read by a CDN and browser.

Prerequisites: complete Exercise A1.

- The following instructions assume you are using Google Chrome. Other browsers have similar features.

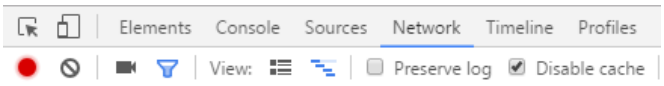
1. Open the **AlloyDemo** project.
2. In Visual Studio, ensure that **Google Chrome** is set as the default browser for running your website.



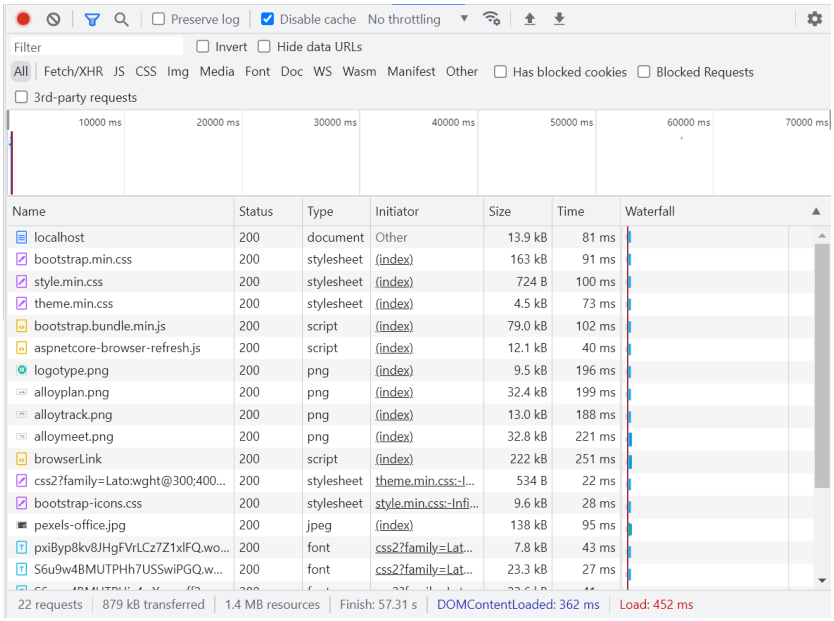
3. Start the website project.

- Make sure that you are NOT logged in. Output caching is only enabled for anonymous users!

4. In **Google Chrome**, press *F12* to show the developer tools pane.
5. Click the **Network** tab.
6. Check the **Disable cache** check box to ensure requests will not be read from the browser's local cache, as shown in the following screenshot:



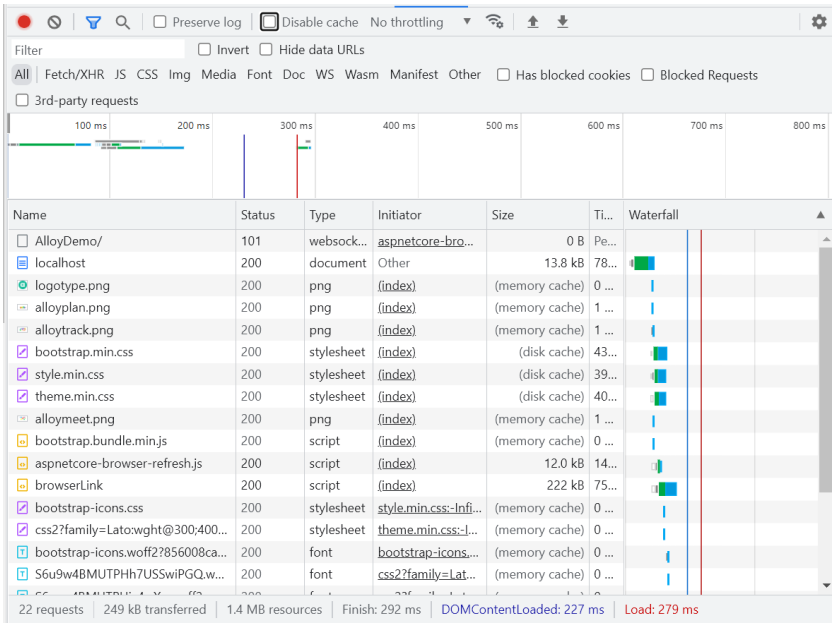
7. Press *F5* to refresh the site's Start page. You will see all the HTTP requests recorded in the developer tools pane, as shown in the following screenshot:



- Note that the Status code returned is **200** OK for all requests. Note the load time was 452 ms and 879 KB was transferred over 22 requests for resources.

8. Uncheck the **Disable cache** check box to allow requests to be read from the browser's local cache.

9. Press *F5* to make the same request, and note the differences in the **Size** column, as shown in the following screenshot:



- Note that the Status code returned is **200** OK for all requests. Note the load time was 279 ms and 249 KB was transferred. Only the request for **localhost** was sent from the web server, all other requests were handled from either disk cache (for stylesheets and scripts) or memory cache (for images) in the browser.

Viewing the HTTP headers for three types of response

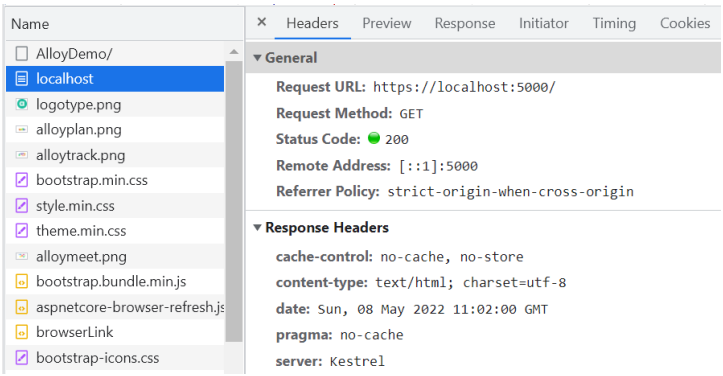
The resources used on the Start page can be divided into three types:

- **Dynamically-generated content**: for example, **localhost**.
- **Static application files**: for example, bootstrap.css, style.css, bootstrap-responsive.css, media.css, editmode.css, jquery.js, bootstrap.js, glyphicons-halflings.png, searchbuttonssmall.png
- **Static asset content**: for example, logotype.png, alloymeetbanner.png, alloyplan.png, alloymeet.png, alloytrack.png

By default, each of the three types is treated differently for caching purposes. This is something you will likely want to configure to optimize for your site.

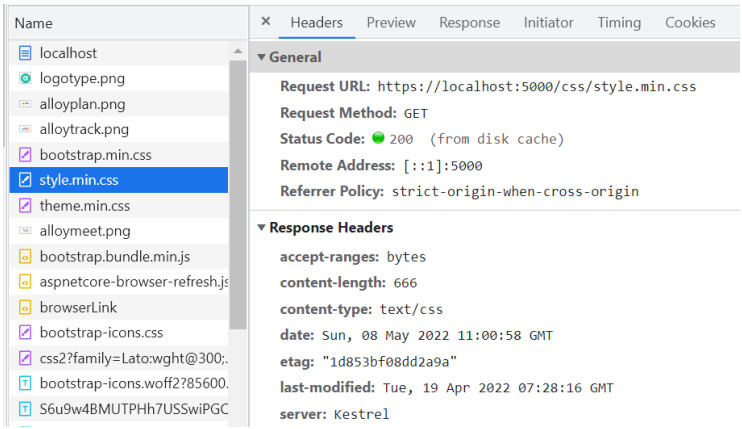
1. In the list of requests, click the first one, named **localhost**, and ensure that the **Headers** tab is selected. This response is dynamically generated at runtime from an MVC view. Note the **Cache-**

Control header is set to **no-cache, no-store**, and there is no **Expires** value after the **Date** header, so the browser won't cache it, as shown in the following screenshot:



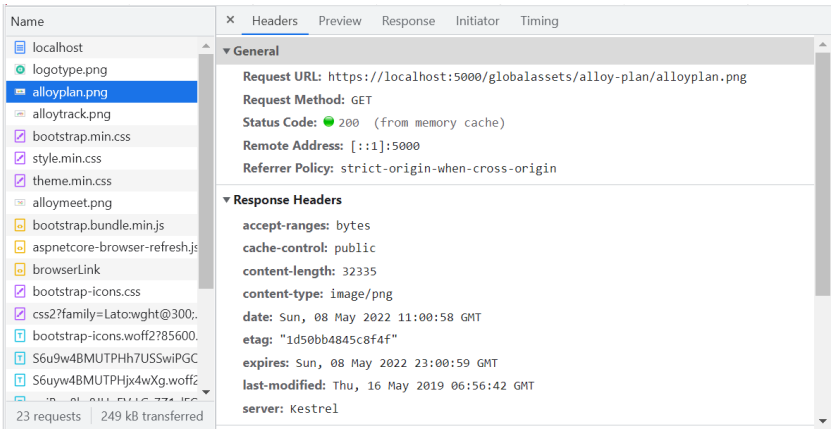
- Good practice would be to change the caching to public and consider if and for how long to cache based on the frequency that the dynamically-generated content changes.

2. In the list of requests, click the one named **style.min.css**, and ensure that the **Headers** tab is selected. This response is static and loaded from the web server's file system, as shown in the following screenshot:



3. In the list of requests, click the one named **alloyplan.png**, and ensure that the **Headers** tab is selected. Note the **Cache-Control** header is set to **public**, meaning that the browser and any

intermediaries can store it until it expires. Its **Expires** header is 12 hours after the **Date** header, as shown in the following screenshot:



4. Close the browser and shut down the web server.

Controlling caching of dynamically generated content with code

For total control of how dynamically generated content is cached, use code.

1. In **Startup.cs**, import namespaces to work with HTTP headers, as shown in the following code:

```
using Microsoft.AspNetCore.Http; // GetTypedHeaders()
using Microsoft.Net.Http.Headers; // CacheControlHeaderValue, HeaderNames
```

2. In the `ConfigureServices` method, add response caching, as shown in the following code:

```
services.AddResponseCaching();
```

3. In the `Configure` method, use response caching, and configure defaults for the HTTP response headers, as shown in the following code:

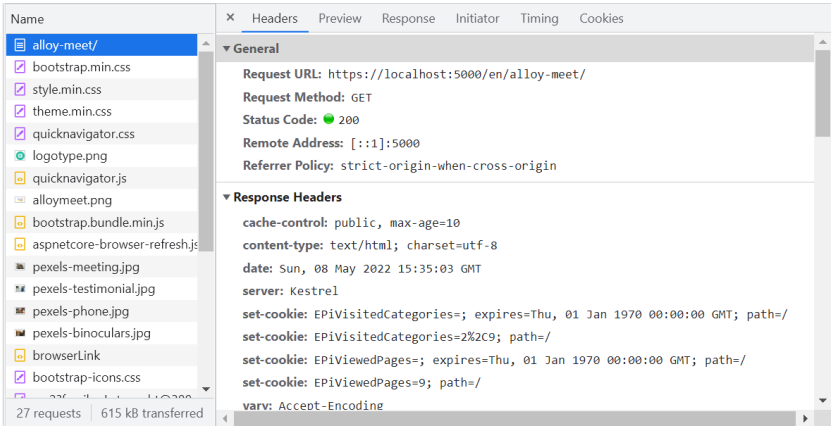
```
app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };

    context.Response.Headers[HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});
```

4. Start the **AlloyDemo** website project.
5. View the recorded network requests in the developer tools pane. Note that the **Cache-Control** header is still set to **no-cache, no-store**.

6. At the command line or terminal, note the warning that the cache control and pragma headers have been overridden because the Start page uses antiforgery, as shown in the following output:

```
warn: Microsoft.AspNetCore.Antiforgery.DefaultAntiforgery[8]

      The 'Cache-Control' and 'Pragma' headers have been overridden and set
to 'no-cache, no-store' and 'no-cache' respectively to prevent caching of
this response. Any response that uses antiforgery should not be cached.
```

7. Navigate to Alloy Meet and view the recorded network requests in the developer tools pane. Note that the **Cache-Control** header is set to **public**, meaning that both the browser *and* CDNs can cache it, the **max-age** is set to 10 (seconds), and there is an **vary** header to ensure that different response encodings are cached separately, as shown in the following screenshot:



8. Close the browser and shut down the web server.
9. To avoid confusion in later exercises, in **Startup.cs**, comment out the three statements that configure response caching.

Controlling caching of dynamic content with attributes and configuration

A simpler, but less flexible and powerful, alternative to writing code is to use a combination of configuration and attributes.

1. In the **Controllers** folder, open **DefaultPageController.cs**.

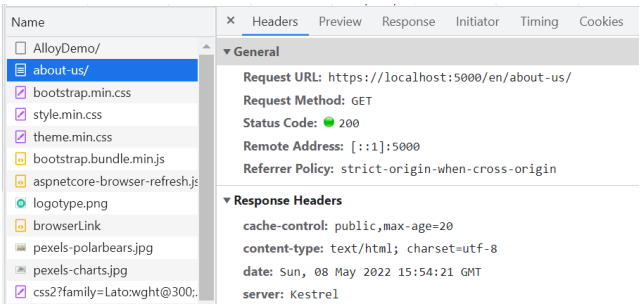
- This controller is used for all page types in Alloy that do not have their own specific controller.

2. Apply the `[ResponseCache]` attribute *before* the **Index** method:

```
[ResponseCache(Duration = 20, Location = ResponseCacheLocation.Any)]
public ViewResult Index(SitePageData currentPage)
```

3. Start the **AlloyDemo** website project.

4. View the recorded network requests in the developer tools pane for the **About us** page and note the HTTP response headers have been affected, for example, **max-age** is **20** seconds, as shown in the following screenshot:



5. Close the browser and shut down the web server.
6. To avoid confusion in later exercises, open **Controllers\DefaultPageController.cs**, and comment out the `[ResponseCache]` that enabled caching.

   Common **max-age** values are:

   - One minute: max-age=60
   - One hour: max-age=3600
   - One day: max-age=86400
   - One week: max-age=604800
   - One month: max-age=2628000
   - One year: max-age=31536000

Controlling caching of objects

1. In **appsettings.json**, add the **EPiServer** | **Cms** | **Content** element to set the expiration to 4 hours (12 hours is the default) for any content loaded by the CMS like pages, blocks, and media asset metadata, as shown in the following configuration:

```
{
  ...
  "EPiServer": {
    "Cms": {
      ...
      "Content": {
        "ContentCacheSlidingExpiration": "04:00:00"
      }
    }
  }
}
```

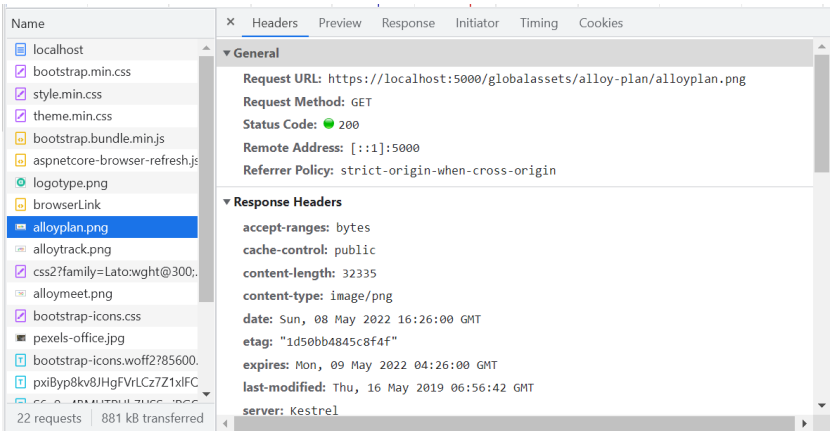
Controlling caching of static content like image assets

1. Start the **AlloyDemo** website project.

- Make sure that you are NOT logged in. Static content asset output caching is set to **Public** for anonymous users, but **Private** for authenticated users by default.

2. Show the developer tools by pressing *F12*.
3. Press and hold the Refresh button, and then choose **Empty cache and hard reload**.

4. View the recorded network requests in the developer tools pane for the **alloyplan.png** image on the Start page and note the HTTP response headers have been affected. The difference between the **Date** and **Expires** is 12 hours, as shown in the following screenshot:



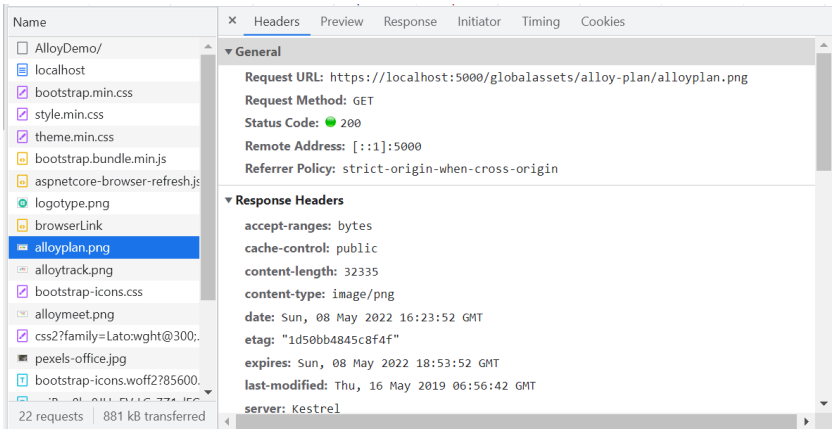
5. Close the browser and shut down the web server.
6. In **appsettings.json**, add the **EPiServer** | **Cms** | **Media** element to set the expiration to 2½ hours for any static media assets managed by the CMS, such as images, videos, and so on, as shown in the following configuration:

```
{
  ...
  "EPiServer": {
    "Cms": {
      ...
      "Media": {
        "ExpirationTime": "02:30:00"
      }
    }
  }
}
```

- In this exercise, you set the expiration to 2½ hours, but a year would be better practice in real life.

7. Start the **AlloyDemo** website project.
8. Show the developer tools by pressing *F12*.
9. Press and hold the Refresh button, and then choose **Empty cache and hard reload**.

10. View the recorded network requests in the developer tools pane for the **alloyplan.png** image on the Start page and note the HTTP response headers have been affected. The difference between the **Date** and **Expires** will now be 2½ hours instead of 12 hours, as shown in the following screenshot:



11. Close the browser and shut down the web server.

Controlling caching of static application files

Static application files like stylesheets and JavaScript libraries are controlled by the web server rather than the CMS:

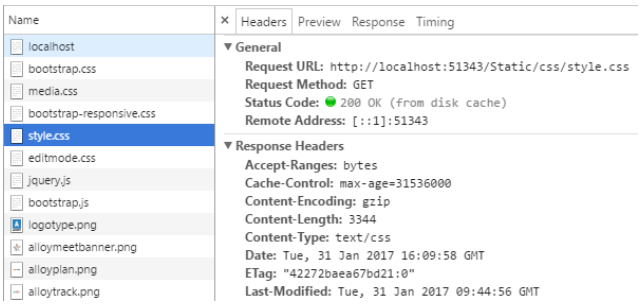
1. In `Startup.cs`, in the `Configure` method, modify the statement that calls `UseStaticFiles` to set to **UseMaxAge** and a **cacheControlMaxAge** set to a time span of 1 day, as shown in the following code

```
app.UseStaticFiles(new StaticFileOptions
{
 OnPrepareResponse = ctx =>
 {
  // seconds_per_minute * minutes_per_hour
  // * hours_per_day * days_per_year
  const int seconds = 60 * 60 * 24 * 365;

  ctx.Context.Response.Headers[HeaderNames.CacheControl] =
   $"public,max-age={seconds}";
 }
});
```

- When you set a “far future” max-age value for static application resources, you should include a version number or date as part of the filename, e.g. change **style.css** to **style-2017-01-31.css** or **jquery.js** to **jquery-3.1.1.js**.

2. Start the **AlloyDemo** website project.
3. Show the developer tools by pressing *F12*.
4. Check the **Disable cache** check box, and then press *F5* to refresh the previously cached for 1 day static application files.
5. Uncheck the **Disable cache** check box, and then press *F5*.
6. View the recorded network requests in the developer tools pane for the **style.css** file on the Start page and check that the HTTP response headers have been affected. The **max-age** is now

31536000 seconds (one year) instead of 86400 seconds (one day), as shown in the following screenshot:



7. Close the browser and shut down the web server.

Exercise G2 – Obscuring a CMS site

In this exercise, you will change the CMS URL path to implement security through obscurity. You will replace the standard known relative paths with alternatives, as shown in the following list:

- Replace `/episerver/cms` with `/secret/cms`
- Replace `/episerver/cms/admin` with `/secret/cms/admin`

Prerequisites: complete Exercise A1.

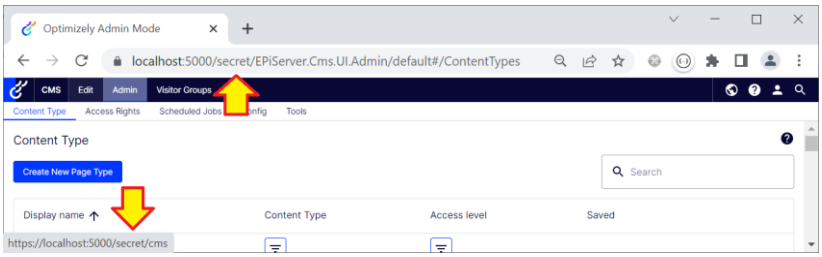
Changing the CMS URL path

You will now modify the site's configuration to change the URL paths.

1. In `Startup.cs`, in the `ConfigureServices` method, add statements to configure the paths to access the CMS user interface and utility functions like registering an admin account, as shown in the following code:

```
services.Configure<EPiServer.Shell.Modules
   .ProtectedModuleOptions>(options =>
{
   options.RootPath = "~/secret";
});

services.Configure<EPiServer.Web.UIOptions>(options =>
{
   options.EditUrl = new Uri("~/secret/cms", UriKind.Relative);
});
```

2. Start the site, and log in as **Admin**.
3. Navigate to Edit view, and note all the menus in the top menu should now use **Secret** as the URL path, as shown in the following screenshot:



4. Close the browser and shut down the web server.
5. In `Startup.cs`, comment out the statements that changed the URLs (or change them to `optimizely` instead of `episerver`!).
6. Start the website and confirm that the path is back to `episerver/cms` as normal.

Exercise G3 – Configuring multi-site

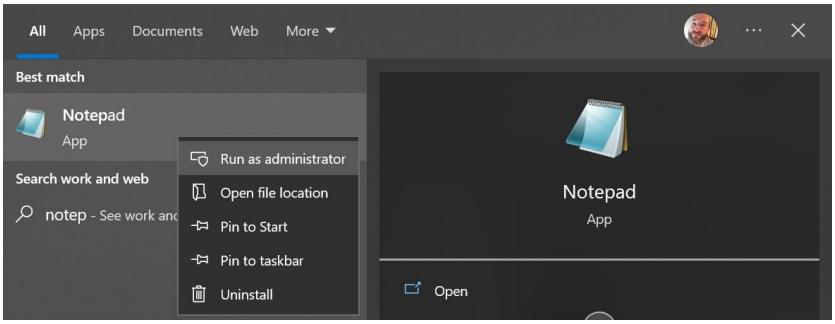
In this exercise, you will implement two sites, each with their own start page, using the multi-site feature.

Prerequisites: complete Exercise A1.

Mapping host names to the loopback address for development

First, we must modify a Windows system file that affects the TCP/IP driver:

1. Run **Notepad** as an administrator, as shown in the following screenshot:



2. Navigate to **File** | **Open** or press *Ctrl* + *O*.
3. Navigate to the `C:\Windows\System32\drivers\etc` folder, filter **All Files (*.*)**, select the `hosts` file, and click **Open**.
4. Add entries for Alloy to map five host names to the IP address of the loopback address on your local computer, as shown in the following text:

```
# BEGIN Optimizely Content Cloud Development Fundamentals training
# The following five entries are used in an exercise.
127.0.0.1          www.alloy.com
127.0.0.1          alloy.local
127.0.0.1          careers.alloy.com
127.0.0.1          www.alloy.se
127.0.0.1          www.alloy.dk
# END Optimizely Content Cloud Development Fundamentals training
```

5. Save changes and close the file.

Configuring sites for Kestrel using port 801

If you have IIS installed, then you could set up an IIS site and use port 80. We want to use the cross-platform Kestrel because that is what is used when deploying to Optimizely DXP cloud hosting along with Linux in a container. If IIS is already listening on port 80, thenwe must use a different port number for the Kestrel web server, like 801, and that port will need to be specified when making requests in the browser.

1. In the **AlloyDemo** project, in **Program.cs**, after the call to `UseStartup`, add a call to `ConfigureWebHost` and `ConfigureKestrel`, as shown in the following code:

```
namespace AlloyDemo;

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

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
```

```
                .ConfigureCmsDefaults()
                .ConfigureWebHostDefaults(webBuilder =>
                    webBuilder.UseStartup<Startup>())

                // tell Kestrel to listen on 127.0.0.1 for requests to
                // port 801 e.g. http://www.alloy.com:801/
                .ConfigureWebHost(options =>
                {
                    options.ConfigureKestrel(options =>
                    {
                        options.Listen(System.Net.IPAddress.Loopback, 801);
                    });
                });
    }
```

- To configure HTTPS, you will need to register a certificate. You can learn how at the following link: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/endpoints

2. In the **AlloyDemo** project, in the **Properties** folder, in **launchSettings.json**, modify the **applicationUrl** setting to use an unencrypted URL to `www.alloy.com` on port 801, as shown in the following configuration:

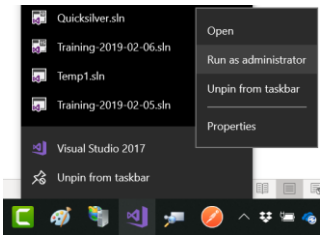
```
{
  "profiles": {
    "AlloyDemo": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "http://www.alloy.com:801/",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

3. Save changes and close.

Managing CMS websites and languages using Admin view

Now we can start the website project and use Admin view to do the rest of the needed configuration to have multiple sites/start pages:

1. Run **Visual Studio** as an administrator, as shown in the following screenshot:



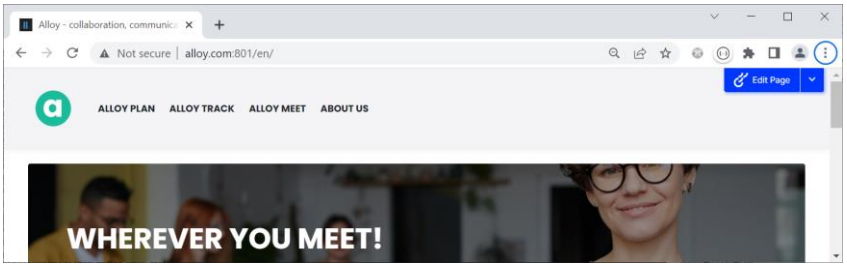
- You need to start Visual Studio with elevated permissions so that IIS Express can be configured correctly.

2. Open the solution with the **AlloyDemo** project.

3. Start the **AlloyDemo** website, and at the command line or terminal, note that Kestrel is configured to use the `alloy.com` domain, as shown in the following output:

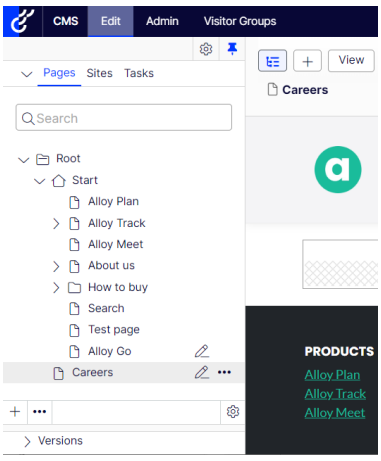
```
warn: Microsoft.AspNetCore.Server.Kestrel[0]
      Overriding address(es) 'http://www.alloy.com:801/'. Binding to
endpoints defined via IConfiguration and/or UseKestrel() instead.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:801
```

4. In Chrome, note the URL, as shown in the following screenshot:



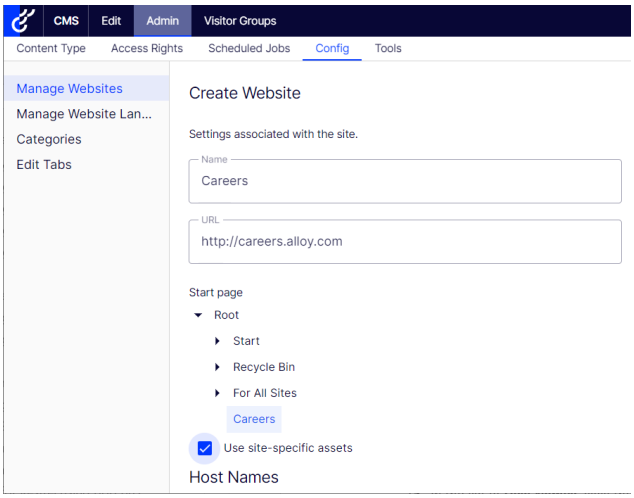
- If you used IIS instead of Kestrl, or if you do not have IIS enabled so Kestrel could be configured to use port 80, then the only difference would be the unnecessary `:801` and the URL would look exactly as a real visitor would see it.

5. Log in as **Admin**.
6. Navigate to **CMS** | **Edit**.
7. Add a new **Start** page under **Root** named **Careers**, as shown in the following screenshot:

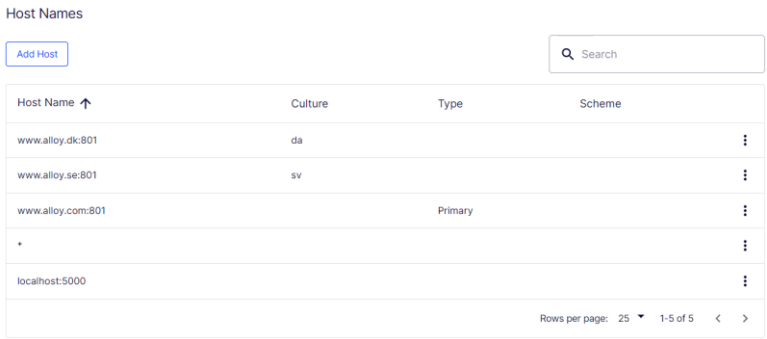


8. Publish the **Careers** page.
9. Navigate to **CMS** | **Admin** | **Config** | **Manage Websites**.
10. Click the **Create Website** button, and provide information to define a new site for careers at Alloy, as shown in the following screenshot:
    a. Name: **Careers**
    b. URL: **http://careers.alloy.com:801/**
    c. Start page: select the **Careers** page that you just created.

d. Use site-specific assets: checked

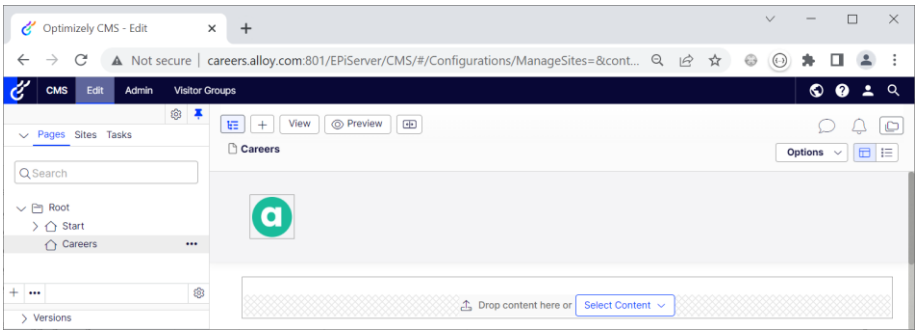


11. Click **Create Website**.
12. In the list of **Websites**, click **AlloyDemo**.
13. Change the **URL** to: **http://www.alloy.com:801/**
14. In the list of **Host Names**, click the **Add Host** button, enter the following values, and then click the Save button:
    a. Host Name: **www.alloy.dk:801**
    b. Culture: **da**
15. In the list of **Host Names**, click the **+ Add** button, enter the following values, and then click the Save button:
    a. Host Name: **www.alloy.se:801**
    b. Culture: **sv**
16. Click **Save Website**.
17. Click **AlloyDemo** and review the host names table, as shown in the following screenshot:

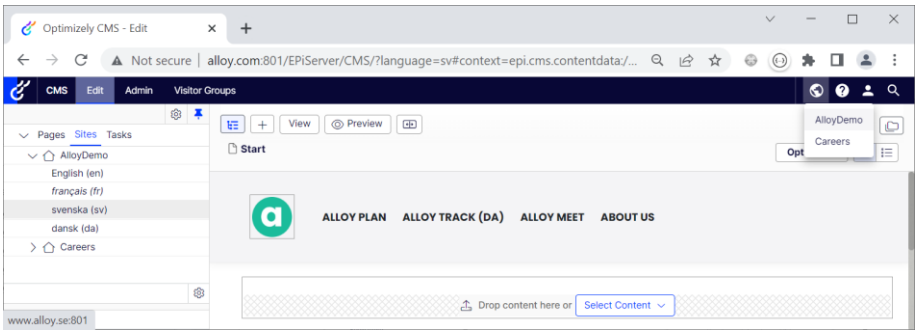


18. Navigate to **CMS** | **Edit**.

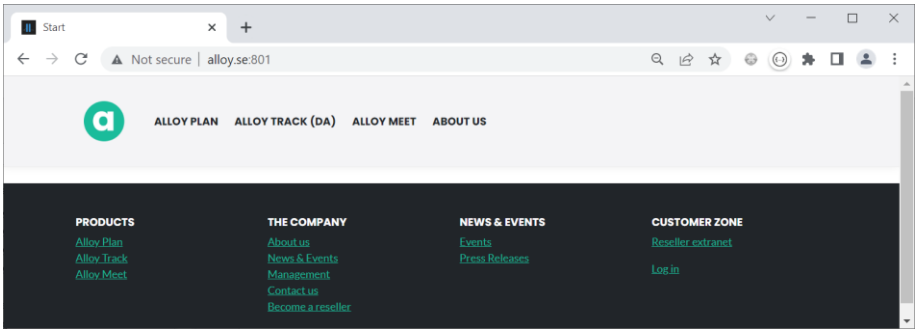
19. Edit the **Careers** page, and note that it is a new site with a house icon and the browser address shows `careers.alloy.com`, as shown in the following screenshot:



20. Edit the **Start** page.
21. Switch to the Swedish language branch by selecting **svenska (sv)** on the **Sites** tab, and then click the globe icon and then **AlloyDemo** to switch to **Live** view, as shown in the following screenshot:



22. You should now see the visitor view of the Swedish language branch of the Start page using the customized host domain **.se**, as shown in the following screenshot:



23. Close the browser and shut down the web server.

- This is the end of the exercises.

Appendix

Summary of Attributes in CMS

Attributes for Content Types

Name	Parameters	Description

`ContentType`	`DisplayName`
	`Description`
	`GroupName`
	`Order`
	`GUID`
	`AvailableInEditMode`
		Controls the registration of a content type in the CMS database.

`Access`	`Roles`
	`Users`
	...others
		Controls who can create an instance of this type.

`ImageUrl`	`Path`	Sets a 120 x 90 preview icon.

`AvailableContentTypes`	`Availability`
	`Include`
	`IncludeOn`
	`Exclude`
	`ExcludeOn`
		Controls which content types can be this content type's parent and children.

`MediaDescriptor`	`Extensions` `ExtensionString`	Controls which file extensions are associated with this media type.

Attributes for Properties

Name	Parameters	Description

`Display`	`Name`
	`Description`
	`GroupName`
	`Order`
		Controls the registration of the property in the CMS database.

`AllowedTypes`	`AllowedTypes`
	`RestrictedTypes`
	...others
		Controls which content types can be referenced by the property.

`CultureSpecific`		Allows the property to have language branches.
`Searchable`		Includes the property in the search index.
`Editable`	`allowEdit` `AllowInitialValue`	Boolean control over editability.
`ReadOnly`		Makes the property readonly.
`Ignore`	n/a	Prevents property from being stored in CMS database.
`ScaffoldColumn`	`false`	Hides the property from Editors.
`UIHint`	`uiHint` `presentationLayer`	For visitors: selects a DisplayTemplate. For editors: selects a custom property editor.
`SelectOne`	`SelectionFactory`	Editors can select one value from a drop-down listbox.
`SelectMany`	`SelectionFactory`	Editors can select many values from a list of check boxes.
`Required`, `Range`, `StringLength`, `RegularExpression`, and so on	...various	Validation rules are applied when saving and publishing property values.

Attributes for Extensions

Name	Parameters	Description

`EditorDescriptorRegistration`	`TargetType`
	`UIHint`
	`EditorDescriptorBehavior`
		Registers a class as a custom editor for a data type.

`UIDescriptorRegistration`	`n/a`	Registers a class to customize UI elements for a content type.

`Component`	`Title`
	`Description`
	`PlugInAreas`
	`SortOrder`
	`AllowedRoles`
	`Categories`
		Registers a class as a gadget for an Editor or Admin to add to their Dashboard or Edit view Navigation and Assets panes.

`GuiPlugIn`	`Area`
	`Category`
	`SortIndex`
	`DisplayName`
	`Description`
	`DefaultEnabled`
	`RequiredAccess`
		Registers a class as a plug-in. `Category` is only used by Visitor Group Criteria.

`ScheduledPlugIn`	`DisplayName`
	`Description`
	`InitialTime`
	`IntervalLength`
	`IntervalType`
	`Restartable`
	`SortIndex`
	`GUID`
		Registers a class as a scheduled job. It can optionally inherit from `ScheduleJobBase` or have a static method named `Execute` that returns a string.

Attributes for Controllers

Name	Parameters	Description

`TemplateDescriptor`	`Name`
	`Description`
	`Inherited`
	`Default`
	`Tags, TagString`
	`AvailableWithoutTag`
	`ModelType`
	`Path`
	`TemplateTypeCategory`
		Controls the registration of a content template in the CMS database.

Attributes for Groups and Tabs

Name	Parameters	Description
`GroupDefinitions`	`n/a`	Apply to a static class with string constants to define tabs in code.
`Display`	`Name` `Order`	Controls the registration of the tab names in the CMS database.
`RequiredAccess`	`AccessLevel`	Controls what access rights an Editor must have to see a tab and its properties.

Attributes for Initialization Modules

Name	Parameters	Description
`InitializableModule`	`UninitializeOnShutdown`	Apply to a class to register it as an initialization module. It must implement

		`IInitializableModule` or `IConfigurableModule`.
`ModuleDependency`	One or more types.	Controls the order in which initialization modules execute by defining dependencies.

Blog Articles about Optimizely CMS 12

Use built in backing typed properties for Optimizely CMS 12

By Eric Petersson

“If you have been looking for some default built in properties for extending your editor's needs such as choosing from weekdays or page types, search no more - this post has got you covered!”

https://ericceric.com/2021/11/29/use-default-backing-typed-properties-for-optimizely-cms-12/

Upgrading Episerver CMS 11 to Optimizely CMS 12, porting to .net 5

By Luc Gosso

“Time to upgrade to .net 5? This is a first ever blog journey going from old .netFramework 4.8 to .net5 and CMS 11 to 12. Easy peasy? Not really, but this guide will halfen your problems.”

https://devblog.gosso.se/2021/11/upgrading-episerver-cms-11-to-optimizely-cms-12-porting-to-net-5/

Cache Tag Helpers in Optimizely 12

By FRANCISCO QUINTANILLA

“In this blog post, I’ll share some code that I used while I was testing this new version. Those code snippets are self-explanatory and it will be easy to transport them to your projects. This was inspired by Scott Reed during the master class “Preparing for Optimizely CMS 12 and Commerce 14”.”

https://powerbuilder.home.blog/2021/11/26/cache-tag-helpers-in-optimizely-12/

Secret debug tools in Optimizely CMS 12

By Allan Thraen

“Optimizely CMS 12 comes with a couple of hidden debug gems.”

https://www.codeart.dk/blog/2021/11/secret-debug-tools-in-optimizely-cms-12/

Listing all endpoints in Optimizely CMS 12 / .NET 5

By Allan Thraen

“Routing has significantly changed in .NET 5 - and that affects many parts of Optimizely (Episerver) CMS 12. For example we have to get used to endpoints a middleware.”

https://www.codeart.dk/blog/2021/11/listing-all-endpoints-in-optimizely-cms-12--.net-5/

Optimizely CMS 12 – .NET 5.0 – Build/Deploy in DevOps (YAML)

By Eric Markson

“The second I created a NET 5 project, the first anxiety-filled thought that ran through my head was something along the lines of… “How the heck am I going to build/deploy this to anywhere useful?!””

https://optimizelyvisuals.dev/2021/11/optimizely-cms-12-net-5-0-build-deploy-in-devops-yaml/

Adding Secure Headers in Optimizely CMS 12

By Stotty

“In any website build it is best practice to remove any headers that your website may produce which expose the underlying technology stack and version. It is also best practice to provide headers which instruct the user's browser as to how your website can use third parties and be used by third parties in order to offer the best protection for the user.”

https://world.optimizely.com/blogs/mark-stott/dates/2021/11/adding-secure-headers-in-optimizely-cms-12/

Restore Root CMS episerver Login Path In .NET 5

By Scott Reed

"With the release of CMS 12 there was the removal of dashboards which has caused the base login path to not work ... Now with the out of the box code you have to navigate to /episerver/cms to access the CMS Editing area."

https://world.optimizely.com/blogs/scott-reed/dates/2021/11/restore-root-episerver-login-path-in--net-5/

Run Once Migration Step For Use With Commerce 14

By Scott Reed

"As part of the release of the new .NET 5 version of the platform, Commerce 14 finally got rid of the Commerce Manager needed for the management of some commerce settings. Instead, now we are presented with a number of features that have been added to the standard Commerce Interface when logging in to your main web application. However, as part of this process a number of areas that were previously availible as UI editable settings in Commerce Manager have been removed."

https://world.optimizely.com/blogs/scott-reed/dates/2021/11/run-once-migration-step/

The New Optimizely CMS 12 Project Structure Explained

By Jon D. Jones

"In this tutorial, you will learn a little more about the new Optimizely CMS 12 project structure and how it differs from CMS 11 and below."

https://www.jondjones.com/learn-optimizely/cms/the-new-optimizely-cms-12-project-structure-explained/

Optimizely CMS 12 – .NET 5.0 – Error ID DXCS006

By Eric Markson

"I recently was able to start using one of the new Linux (.NET5.0/CMS12) DXP instances to do some testing/development in, and I came across an interesting error that threw me for a loop."

https://optimizelyvisuals.dev/2021/11/optimizely-cms-12-net-5-0-error-id-dxcs006/

Integrating Azure AD with Optimizely CMS 12

By Bob Davidson

"One area that has seen some relatively large changes is integrating with Azure Active Directory."

https://www.blendinteractive.com/news/integrating-azure-ad-with-optimizely-cms-12/

Creating Your First Page In Optimizely CMS 12

By Jon D. Jones

"Within this tutorial, you will learn how to create your first CMS page within Optimizely 12 (formally Episerver CMS)."

https://www.jondjones.com/learn-optimizely/cms/creating-your-first-page-in-optimizely-cms-12/

AppSettings and AppSettingsMultiple properties for Optimizely CMS 12

By Eric Petersson

"I wanted to find out if there were some kind of new properties to foul around with for the CMS 12 design pattern. And there were in some kind of regard some new ones to be unveiled 🙋"

https://ericceric.com/2021/11/05/appsettings-and-appsettingsmultiple-properties-for-optimizely-cms-12/

New Geta.NotFoundHandler

By Māris Krivtežs

"BVN.404Handler and Geta.404Handler are re-released as Geta.NotFoundHandler. The new library supports ASP.NET 5 and Optimizely 12."

https://marisks.net/2021/11/04/new-geta-notfoundhandler/

Use Deployment API to Deploy a Hotfix in Optimizely/Episerver DXP

By Jorge Cardenas

"If for some reason you need to deploy a hotfix to any environment in DXP sometimes is annoying to use the paasportal to move from the code to one environment to another. The solution is to use the deployment API provided by Optimizely which allows you to make this kind of actions."

https://www.cdisol.blog/2021/10/30/use-deployment-api-to-deploy-a-hotfix-in-optimizely-episerver-dxp/

Running Optimizely 12 on a Mac

By Andy Blyth

"Now that Optimizely 12 runs on .NET 5 it can be run on a Mac, and with most front enders using Macs, I thought it might be useful to demonstrate the steps on how to get it up and running on a Mac."

https://www.technicaldogsbody.com/blog/running-optimizely-12-on-a-mac

Working with docker and containers - .NET 5 Series, Part 3

By Mark Hall

"In this post we will explore creating docker images of Optimizely applications and running them in containers."

https://world.optimizely.com/blogs/Mark-Hall/Dates/2021/9/working-with-docker-and-containers----net-5-series-part-3/

Starting a new project - .NET 5 Series, Part 2

By Mark Hall

"In this post we will explore starting a new project."

https://world.optimizely.com/blogs/Mark-Hall/Dates/2021/8/starting-a-new-project----net-5-series-part-2/

Getting started with upgrade-assistant - .NET 5 Series, Part 1

By Mark Hall

"In this post we will explore the upgrade-assistant tool from Microsoft. This tool allows for developers to upgrade their .net full framework applications to .NET 5 and beyond. It is also built to be extensible so ISV's can add extensions to make it easier to for their customers to upgrade their solutions. Optimizely has created its own extension library to automate common Optimizely specific fixes when moving to .NET 5 and beyond."

https://world.optimizely.com/blogs/Mark-Hall/Dates/2021/7/getting-started-with-upgrade-assistant----net-5-series-part-1/

Audit logging add-on for netcore-preview

By Antti Alasvuo

"Roughly a year ago I created the Swapcode.Episerver.AuditLog package to log access right changes to the built-it activity log (Change log) ... I had some free time, so I decided why not prepare the package for the .NET 5 implementation and at the same time rename it to reflect the new Optimizely name."

https://world.optimizely.com/blogs/Antti-Alasvuo/Dates/2021/9/audit-logging-add-on-for-netcore-preview/

- This is the end of the exercise book.