* **dotnet run** 
  + **start the app**
* **dotnet watch run** 
  + **regenerate the source code and rebuild automatically**

**Umbraco -> Settings -> Models Builder -> Generate Models**

* **This will generate models in the file <root>/umbraco/models**
  + **These represent the document types and media types within umbraco such that we can reference them and get their values from them**

**Document types**

* **Document type with a template**
  + **Upon a creation of one named home.space => that will create a template called home page and it will automatically assign that template to that document type. So when we will create a document item using that document type, it’ll be mapped to that view.**
* **Document Type**
  + **So without a template**
* **Element type**
  + **What we tipically use for repeating lists like for a block list or nested content**
* **Composition**
  + **Common and re-usable properties. Maybe we will have a title and we want that on multiple different pages, we can pick a composition on the pages that we want it on.**
  + **Therefore, we are not adding any pages, we are just pulling it in from where it was originally created.**
* **Names and aliases of properties have to be unique.**
* **Don’t forget to allow the document types upon creating them**
* **On the page content**
  + **We can see the Id and the key.**
  + **The id may be different on different environments but the key will be the same.**
  + **The id is the one umbraco is using in the URL when editing**
* **After creating our documents and templates, we need to generate the models such that their properties are updated**

**Razor**

* **Razor code blocks are denoted by @. C# code can be embedded within HTML using @{ }, or inline with @ before the code expression.**
* **Razor code block start at the @ and finish at the first html element found.** 
  + **So in case we are inside a code block and a html element is specified, we need to use the @ again in order to use another C# piece of code. Which we normally wouldn’t have to since we already were inside a code block.**

**GOOD:**

**A computer screen shot of a code

Description automatically generated**

**WRONG:**

**A screen shot of a computer program

Description automatically generated**

* **If we write C# code outside of code blocks, it will simply be printed on the screen**
* **@<variable-name>**
  + **prints the value of the variable on screen**
* **<list>.any()**
  + **checks list is not empty**

**On umbraco 8, the asset folders for our project would be placed in the root directory of the project.**

**With umbraco 10, all static assets must be placed in the <root>/wwwroot directory, the same folder umbraco puts the media files.**

**Our umbraco project is hosted at**

* **localhost:8467/umbraco**

**or at**

* [**https://localhost:44327/umbraco**](https://localhost:44327/umbraco)

**Templates**

* **Templates are not only created in the files, but they also have a database entry created for them**
* **Partial Views are different, they are only created in the file system.**

**Master template**

* **Selecting a master template for another, will make our template inherit from its master.**
  + **Everything that’s on the master template will be rendered first. From the master, a @RenderBody() method will be called and then our template will be rendered.**
  + **So @RenderBody will render all the content of a page who inherits from its master**
* **The master template will not know about the properties of a model which inherits from it. So we would need to put the properties into a Partial View and let it know about those properties.**
* **Partial View**
  + **naming convention: start with an \_**
  + **We can add code like this** 
    - **@inherits UmbracoViewPage<IheaderProperties>**

**and our partial view will know about it thanks to what we have in the \_ViewImports**

* + **Everything imported into the \_ViewImports file will be made available to all views**

@using Umbraco.Cms.Web.Common.PublishedModels

**which means that we will need to use less @using statements in our views. It will be available to all our views.**

* **Now to use the data from our view in the master template, we can add it using @await Html.PartialAsync(<path-to-view>, Model)**
* **The Model is not mandatory to be passed. It is inferred as the model of whatever is calling it.**
* **Permissions**
  + **We need to allow a page in order for it to be available in our content menu**
  + **We need to allow a page as a child in order for us to place that page under another**
  + **So the children of page A represent all the pages which are placed unde**
* **After editing a certain page, we need to make it inherit from a master such that, it displays the content we need.**
* **Do not forget to add the / to the urls when using the master template such that all children who inherit from the master will have the correct route to the ROOT.**
* **In the main Content page**
  + **the name that is given to the page/component will be the name the route for it will be registered with**
* **Let’s say properties A,B,C are on a view awaited for by the master. Even though the children will inherit from the master and even though the master is @using the published models, we still need to add these properties on the children in order for them to be accessed.**

**So does it really have to be on everyone?**

**ModelsBuilder generates strongly typed models based on your Umbraco document types, ensuring that you have compile-time safety and intellisense support. This means you can access properties and content values with confidence, reducing the risk of runtime errors.**

**UmbracoHelper will expose all content in the form of IPublishedContent. To get a reference to the currently executing content item from the UmbracoHelper, use UmbracoHelper.AssignedContentItem.**

* **umbracoNaviHide is the default alias against which methods like isVisible check.**

**var homePage = Model as ContentModels.HomePage;**

* This means that if the current page is not a HomePage, the homePage variable will be null, and thus the navigation links will not be rendered.
* **Opening a link in a new windo**
  + ****
  + **And select the option from the backoffice**

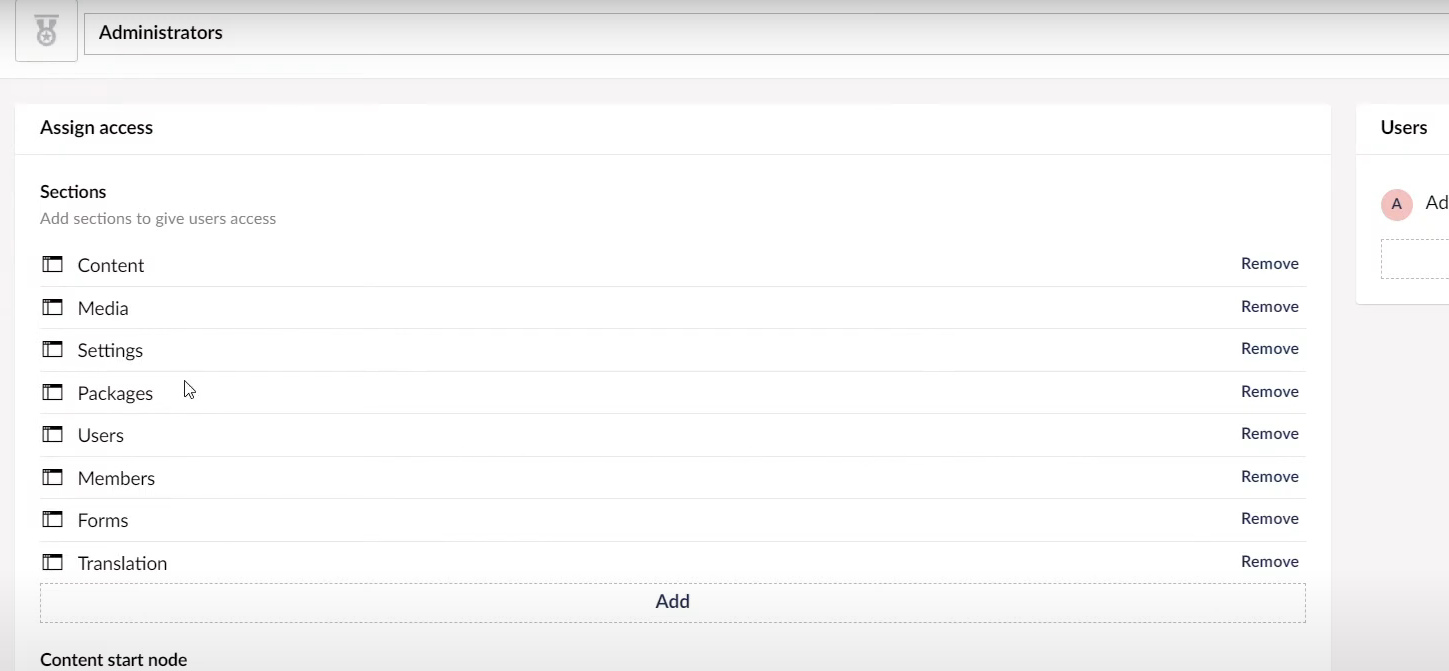
**Part 6 – Nested Content Nav**

* **For nested content or the block list, use the Element Type ( not a document type or a composition)**
* **Defines the schema for a repeating set of properties**
* **How to move the properties from a tab to a group**
* **How to build a custom data type**
  + **Using nested content**

**Part 8 – Footer Icons and Dictionary**

**Users -> Assign Access -> Edit**

* **Granting access to different sections**

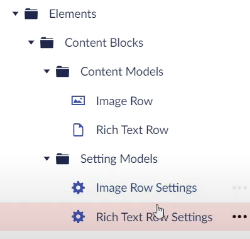
****

* **Access to dictionary values**
  + **@Umbraco.GetDictionaryValue(“<variable-name-from-dictionary>”)**

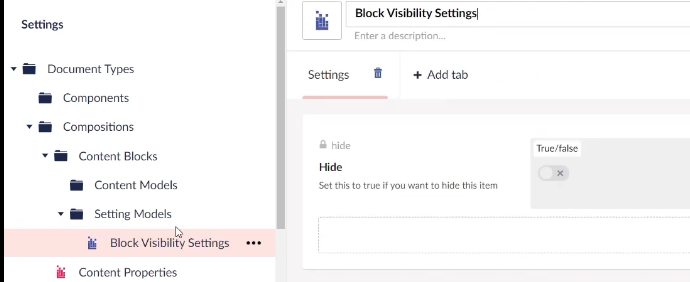
**Part 9 – Block List Editor**

* **BlockList**
  + **A data type with multiple repeatable data types**
* **When defining a block for the blocklist, the block will have**
  + **The actual block for the content**
  + **A block for settings (which is optional)**

1. **Document Types -> Elements -> Content Blocks -> Content Models**
   * **Create an “Element” Type with your desired configuration**
   1. **Document Types -> Elements -> Content Blocks -> Setting Models**

****

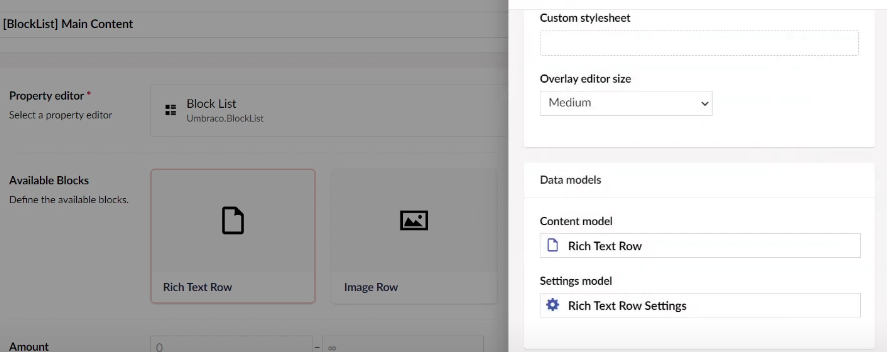
* + **Create an “Element” Type with your desired configuration as settings**
  + **It’s indicated that we create a setting block for each content block we create.**
  1. **Document Types -> Compositions -> Content Blocks -> Content Models + Setting Models**
  + **In the setting Models folder, we create “Composition” Types which will have the properties we want as settings for our content block items.**

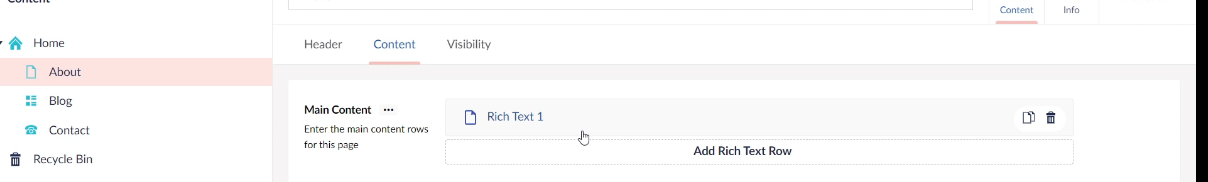
****

* 1. **Now we can add those “Composition” settings on our “Element” settings**

1. **Go to Data Types**
   * **Create an element whose name starts with [BlockList] and then the name**
   * **Select the Block List as the content editor**
   * **“Available Blocks” section will let you add blocks with your created “Element” types**
   * **Labels:** 
     + **{{'Rich Text ' + $index}} {{$settings.hide == '1' ? ' [HIDDEN]' : ''}}**
     + **Denotes how the block will be named when we add it to the content of a page**

**2.1) The settings elements can be added to our block list items in their GUI**

****

****

* + **After that, our block list data type can be added as a property on our pages/compositions etc…**
  + **When editing an element, the setttings tab will also appear in case a settings model has been added.**

1. **In the IDE, go to Views -> Partials -> blocklist** 
   * **Create a folder called “Components”**
     + **Here, we have to create the model file for our block list item**
     + **Tip: Use the default.cshtml file for inspiration. Maybe even add the necessary “using import” to the ViewImports.**
     + **The model should always be named after the alias of the document type we created**
     + **In order to access our element, we have to cast Model.Content to the type of our element.**
     + **In order to access our settings, we have to cast Model.Settings to the type of our settings**

****

1. **Once we have the model of the block list item, we can render it on our page, accessing it like**

**@Html.GetBlockListHtml(Model.<block-list-property-name>**

1. **Whenever we modify our block list by adding new properties, we will get an error since the file under /View/Partials/blocklist has to be created !**

* **Image cropper**

****