|  |  |
| --- | --- |
|  | |
| ASP.NET WEB APP WITH CONTENTFUL HEADLESS CMS  Cooking Tutorials Blog | |
| **Dina Boshnaq 2ACS**  R0823887 | DevOps and Security - IT Factory |
|  |
| Academic Year 2022-2023  Campus Geel, Kleinhoefstraat 4, BE-2440 Geel |



Table of contents

Contents

[Table of contents 3](#_Toc122227827)

[1 Introduction 4](#_Toc122227828)

[2 Corpus 5](#_Toc122227829)

[2.1 Contentful 5](#_Toc122227830)

[2.2 Creating our Solution on VS Code 6](#_Toc122227831)

[2.3 Building the application with Contentful 6](#_Toc122227832)

[2.4 Making The Dockerfile 12](#_Toc122227833)

[2.5 GitHub Actions Pipeline 13](#_Toc122227834)

[2.6 Demo of the project 15](#_Toc122227835)

[3 Sources & References 18](#_Toc122227836)

# Introduction

For this case study, an ASP.NET Application is made about “Cooking Tutorials”. The content of the blog will be retrieved from Contentful which is a Headless Content Management System (CMS). The blog contains 1 introduction page that includes a course menu with the name of the dishes for each course. 3 more pages are added: Appetizers, Main Dishes, and Desserts. Each one of these pages contains cards of the dishes of that course. Each dish card contains the name of the dish, if it’s vegetarian or not (except for desserts), the cooking time, image of the dish, description of the dish, and an embedded tutorial video for making that dish. Programming languages used are C# (since it’s an ASP.NET Core MVC application), HTML, CSS, and JavaScript. The structure of this project is mainly based on Models, Views, and Controllers (MVC).

Along with the application a Dockerfile is made to build the image of the container containing the application. A Github Actions pipeline is made to automate the entire workflow. It builds, tests, and publishes the application. Then it builds the image using the Dockerfile and pushes that image to Docker Hub. The pipeline will also upload an artifact for the application.

In the following pages we will take a deeper look at the steps taken to achieve all that was mentioned above.

**Important Links:**

GitHub Repository: <https://github.com/16dina/caseStudy>

YouTube video: <https://youtu.be/KD56hndjSvE>

# Corpus

## Contentful

An account is made on Contentful. A total of 4 content models are made: Dish Type, Appetizer, Main Dish, Dessert. Each content model has a number of attributes. Different types such as short text, long text, media, Boolean, and integer were used.

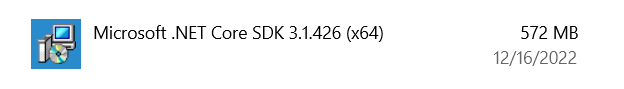
Graphical user interface, text, application

Description automatically generated

|  |  |
| --- | --- |
| Graphical user interface, text, application, email  Description automatically generated | Graphical user interface, text, application, email  Description automatically generated |
| Graphical user interface, text, application, email  Description automatically generated | Graphical user interface, text, application, email  Description automatically generated |

## Creating our Solution on VS Code

To start off we installed the Microsoft .NET Core 3 SDK from the official website.



In our desktop we created a folder called “caseStudy”, here we will be building our app. We open terminal in that folder and run the command **dotnet new mvc**. To start programming we run the command **code .** which will take us to the VS code where our the basic template for and ASP.NET Core MVC application will be setup.

## Building the application with Contentful

For this part we followed a tutorial online from Contentful’s official site. We download the package for Contentful in our solution using NuGet package manager (which also needed to be installed). The command: **Install-Package contentful.aspnetcore**

Then we configure our keys for Contentful in the **appsettings.json** file. These keys are necessary to be able to access the Contentful data. They are found in *Settings*🡪*API* keys in Contentful. For this application we will be needing the Content Delivery API (CDA), but not the Content Management API (CMA) as there was no need for it.

Text

Description automatically generated

Inside the **Startup.cs** file we also have to say that we’re using Contentful by adding the following line **using Contentful.AspNetCore;** and register a service to add and connect with Contentful using the API Keys (configuration) specified before.

Text

Description automatically generated

In the controller **HomeController.cs** we also need to specify that we’re using Contentful but this time we just say **Core**, so **using Contentful.Core;**

We then configure the Contentful client as follows:

Text

Description automatically generated

The next step is to make the models, each model with its attributes and their types:

Graphical user interface, text, application

Description automatically generated

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated |
| Text  Description automatically generated | Text  Description automatically generated |

There’s also error but that comes by default so we won’t be mentioning it in our explanations.

We make asynchronous calls to retrieve the data from our Contentful Models. In this call we use the client with **await** (cause it’s asynchronous) and the method **GetEntries** which allows us to specify an object that will be populated with the data. We store the object in a variable. Finally we pass the variable containing the model object to the corresponding View. We do this for every view. In our case, we are passing 1 model only each view. The following are our pages’ views:

Text

Description automatically generated

Now for the Views. In each view we specify the Contentful Collection we will be using inside the page in order for us to be able to access the attributes of its model. We also specify that we are using the Models so our application knows that this collection is a model. To achieve this we add the following lines in each view with there corresponding model:

**@using Contentful.Core.Models**

**@model ContentfulCollection<***ModelName***>**

The views here are going to be 2 types, shared and individual. The shared view called **\_Layout.cshtml** and **\_Layout.cshtml.css** is going to contain the shared parts in the pages like the navbar and some styling. As for the individual ones they are specific to their own pages.

Text, application

Description automatically generated

Inside of each view we iterate over the model to show all the content of that type. We will look at some parts of 2 of these views, **index.cshtml** and **appetizers.cshtml** so we can also take a look at the JavaScript used in them.

1. Index.cshtml

Text

Description automatically generated

As we see here, one big template is made for a dish type card, and we use the **foreach** to grab all the dish types along with their attributes from Contentful and place each one in a separate card. An addition we can see here is the use of JavaScript to make a menu like tab. If clicked on a button, the dishes of that dish type are shown and the rest are hidden. It looks as follows:

A picture containing logo

Description automatically generated

1. Appetizers.cshtml

Text

Description automatically generated

For the Appetizers’ view (similar to the main and desserts), the same thing is done where we iterate over all the content of the model Appetizers and we make cards for each dish. Here we also used JavaScript in order to figure out whether the dish is vegetarian or not. Inside of the Appetizers model there is an attribute called “isVegetarian” of the type Boolean. If it’s true, we show a vegetarian tag on the dish, otherwise we show a non-vegetarian tag. We include the name description, image, and video tutorial (embedded YouTube video) of the dish as well. It looks as follows:

Graphical user interface

Description automatically generated

Or in the case of non-vegetarian:

Graphical user interface

Description automatically generated

## Making The Dockerfile

Up till now we have been building our application and running it using the commands **dotnet build** and **dotnet run**. But as we have finished setting up the project and making all our views and models, we can now put it in a Docker container and run that container to start everything. We use a Docker image which will act as the blueprint for building the container. We will build this image using a Dockerfile. The following is our Dockerfile:

Text

Description automatically generated

Let’s explain what each part it does:

1. **Line 2:** The base image which is the dotnet core sdk image for .NET 3.1 and we define it as a variable called CS-build
2. **Line 3:** We create a working directory called “app” where we build the project in the container.
3. **Line 6-7:** Copy the csproj file which is in our case the caseStudy.csproj  from our PC to the container. We a resolve any dependencies that we installed using the NUGET package manager previously.
4. **Line 10-11:** We copy the rest of the files to our container into the working directory. We run dotnet publish with the configuration flag of the list, and we place the output of that which is basically a compiled project into a folder called out.
5. **Line 14:** Here we use the .net core aspnet to generate a runtime image (so without the .net core SDK as that’s only necessary when building the application). Thus we make a smaller container which is more efficient.
6. **Line 15:** Specifying the working directory again which is “app”.
7. **Line 16:** Exposing port 80 in the container.
8. **Line 17:** A multi part build. Here we combine parts from the previous build steps and put it in our working directory “app”.
9. **Line 18:** Specifying our entry point which is the thing our container is going to do once its starts. Here use the dotnet command to run the caseStudy.dll which is our app.

|  |  |
| --- | --- |
| **An addition:** We also have a file called **.dockerignore**. This file is telling the docker engine to exclude the directories **bin** and **obj** when it builds the image. This makes the docker image smaller and more efficient. | Graphical user interface, text  Description automatically generated |
|  |  |

## GitHub Actions Pipeline

We create a GitHub repository for our project and put all the necessary folders and files in it. A GitHub Actions pipeline has been made to automate our workflow. The pipeline was based on the .NET Action in the marketplace, and then built on using various sources online.

**Through this pipeline we will:**

1. Build our application, test it on multiple .NET versions (2.1 /3.1 /5.0), and publish it.
2. Publish our project as an artifact called caseStudy.
3. Build the Docker image for our project using a Dockerfile.
4. Place the Image on our Docker Hub account after logging into it.

**Some explanation of parts of the file:**

* We name this pipeline “.Net Core” and set the workflow is triggered whenever we push something to our repository, so whenever a commit happens.
* In “jobs” we specify the actions to be made, and for us it’s a “build” action. Build will help us compile the code, run tests, restoring dependencies, etc.
* We specify platform where the job will run, here it is ubuntu:latest.
* Checkout v3 is an action used to check out our repository under $GITHUB\_WORKSPACE, thus allowing our workflow to access it.
* Each job has a name.
* The multiple .net version testing jobs use the predefined action “setup-dotnet”.
* Uploading an artifact uses the action “upload-artifact”.
* “run” runs commands in a job.
* To access our Docker Hub we need the username and password. These we defined in the Actions Secrets on GitHub.

The file can be seen on the following page.

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generated

## Demo of the project

We now have our Docker image on DockerHub and we can build a container to view the blog. We will try doing this in 2 ways: with a docker compose file, and by simply running a **docker run** command.

Via Docker Compose: Here we have a service that specifies the container “cookingapp-aspnet” to run using the image we pushed to Docker Hub. We make port 80 on our PC forward us to the previously exposed port 80 in the container. Once we run **docker compose up -d** the container will be created and will start in the background.

A picture containing graphical user interface

Description automatically generated

OR via running the following command in our command prompt:

**docker run -d --name cooking -p 8000:80 dinaboshnaq/cookingapp-aspnet**

Both will cause our PC to pull the image from Docker Hub to our local machine.

**Results:**

**GitHub Actions workflow:**

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface, text, application

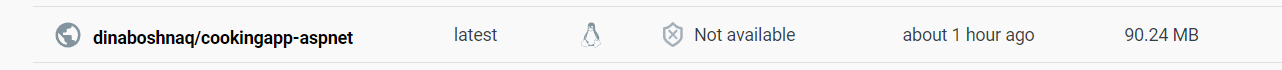
Description automatically generated**

**Upload artifact:**

**Graphical user interface, application

Description automatically generated with medium confidence**

**Image in Remote Repository:**

****

**Image Locally:**

**Graphical user interface, application, Teams

Description automatically generated**

**Containers:**

**Graphical user interface, application

Description automatically generated**

**Surfing to localhost(80):**

**Text

Description automatically generated**

**Surfing to localhost:8000:**

**Text

Description automatically generated**

**Bonus: The favicon is replaced with more suitable one 😊 **

**And this concludes our document.**

# Sources & References

**From YouTube:**

* Code, O. S. A. (2020, May 3). *Using Contentful CMS with ASP.Net Core* [Video]. YouTube. <https://www.youtube.com/watch?v=9EX0uB9dcbM&feature=youtu.be>
* Les Jackson. (2019, August 11). *Deploy a .NET Core API with Docker (Step-by-Step)* [Video]. YouTube. <https://www.youtube.com/watch?v=f0lMGPB10bM>
* Houssem Dellai. (2022, April 15). *Create Github Actions workflow for .NET* [Video]. YouTube. <https://www.youtube.com/watch?v=x5ikUcN2ge4>

**From online pages:**

* *ASP.NET Core tutorial with*. (n.d.). Contentful. <https://www.contentful.com/developers/docs/net/tutorials/aspnet-core/>
* Microsoft (2022, November 18). *Containerize an app with Docker tutorial - .NET*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/docker/build-container?tabs=windowsoft+Learn>
* *W3Schools online HTML editor*. (n.d.). <https://www.w3schools.com/w3css/tryit.asp?filename=tryw3css_templates_pizza>
* Z. (2019, December 30). *Getting Started with Github Actions for .NET Core*. Rapture in Everything. <https://zblesk.net/blog/getting-started-with-github-actions-for-net-core/>
* T. (2022, October 25). *dotnet build command - .NET CLI*. Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-build>
* Ardal, T. (2022, March 1). *Building & testing on .NET versions in GitHub Actions*. elmah.io Blog - .NET Technical Tutorials/Guides and New Features. <https://blog.elmah.io/building-and-testing-on-multiple-net-versions-with-github-actions/>
* Eric L. Anderson. (2020b, June 22). *GitHub: Use Actions to Publish Artifacts*. <https://elanderson.net/2020/06/github-use-actions-to-publish-artifacts/>
* Eric L. Anderson. (2020b, June 16). *GitHub: Use Actions to build ASP.NET Core Application*. <https://elanderson.net/2020/06/github-use-actions-to-build-asp-net-core-application/>

**Other sources include the study material and tips from the teacher.**