How To Deploy an ASP.NET Core Application with MySQL Server Using Nginx on Ubuntu 18.04

Posted July 23, 2019 6.9k DEPLOYMENT APPLICATIONS NGINX MYSQL UBUNTU 18.04

By Oluyemi Olususi
Become an author

The author selected the Open Source Initiative to receive a donation as part of the Write for DOnations program.

Introduction

ASP.NET Core is a high-performant, open-source framework for building modern web applications, meant to be a more modular version of Microsoft's ASP.NET Framework. Released in 2016, it can run on several operating systems such as Linux and macOS. This enables developers to target a particular operating system for development based on design requirements. With ASP.NET Core, a developer can build any kind of web application or service irrespective of the complexity and size. Developers can also make use of Razor pages to create page-focused design working on top of the traditional Model-View-Controller (MVC) pattern.

ASP.NET Core provides the flexibility to integrate with any front-end frameworks to handle client-side logic or consume a web service. You could, for example, build a RESTful API with ASP.NET Core and easily consume it with JavaScript frameworks such as Angular, React, and Vue.js.

In this tutorial you'll set up and deploy a production-ready ASP.NET Core application with

Prerequisites

You will need the following for this tutorial:

up guide,

- Nginx installed by following How To Install Nginx on Ubuntu 18.04.
- A secured Nginx web server. You can follow this tutorial on How To Secure Nginx with Let's Encrypt on Ubuntu 18.04.
- Both of the following DNS records set up for your server. You can follow this introduction to DigitalOcean DNS for details on how to add them.
  - An `A` record with `your-domain` pointing to your server's public IP address.
  - An `A` record with `www.your-domain` pointing to your server's public IP address.
- MySQL installed by following How To Install the Latest MySQL on Ubuntu 18.04.

Step 1 — Installing .NET Core Runtime

A .NET Core runtime is required to successfully run a .NET Core application, so you'll start by installing this to your machine. First, you need to register the Microsoft Key and product repository. After that, you will install the required dependencies.

First, logged in as your new created user, make sure you're in your root directory:

```
$ cd ~
```

Next, run the following command to register the Microsoft key and product repository:

```
$ wget -q https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-pro
```

Use `dpkg` with the `-i` flag to install the specified file:

```
$ sudo add-apt-repository universe
```

Next install the `apt-transport` package to allow the use of repositories accessed via the

```
$ sudo apt install apt-transport-https
```

Now, run the following command to download the packages list from the repositories and update them to get information on the newest versions of packages and their dependencies:

```
$ sudo apt update
```

Finally, you can install the .NET runtime SDK with:

```
$ sudo apt install dotnet-sdk-2.2
```

You will be prompted with the details of the size of additional files that will be installed. Type `Y` and hit `ENTER` to continue.

Now that you're done installing the .NET Core runtime SDK on the server, you are almost ready to download the demo application from GitHub and set up the deployment configuration. But first, you'll create the database for the application.

Step 2 — Creating a MySQL User and Database

In this section, you will create a MySQL server user, create a database for the application, and grant all the necessary privileges for the new user to connect to the database from your application.

To begin, you need to access the MySQL client using the MySQL root account as shown here:

Next, create a MySQL database for the application with:

```
mysql> CREATE DATABASE MovieAppDb;
```

```
Output
Query OK, 1 row affected (0.03 sec)
```

You've now created the database successfully. Next, you will create a new MySQL user, associate them with the newly created database, and grant them all privileges.

Run the following command to create the MySQL user and password. Remember to change the username and password to something more secure:

```
mysql> CREATE USER 'movie-admin'@'localhost' IDENTIFIED BY 'password';
```

You will see the following output:

```
Output
Query OK, 0 rows affected (0.02 sec)
```

To access a database or carry out a specific action on it, a MySQL user needs the appropriate permission. At the moment **movie-admin** does not have the appropriate permission over the application database.

You will change that by running the following command to grant access to **movie-admin** on `MovieAppDb`:

```
mysql> GRANT ALL PRIVILEGES ON MovieAppDb.* TO 'movie-admin'@'localhost';
```

Now, you can reload the grant tables by running the following command to apply the changes that you just made using the flush statement:

You will see the following output:

```
Output
Query OK, 0 rows affected (0.00 sec)
```

You are done creating a new user and granting privileges. To test if you are on track, exit the MySQL client:

```
mysql> quit;
```

Log in again, using the credentials of the MySQL user you just created and enter the appropriate password when prompted:

```
$ mysql -u movie-admin -p
```

Check to be sure that the user **movie-admin** can access the created database, check with:

```
mysql> SHOW DATABASES;
```

You will see the `MovieAppDb` table listed in the output:

```
Output
+--------------------+
| Database           |
mysql> quit;
```

You've created a database, made a new MySQL user for the demo application, and

ıe next

Step 3 — Setting Up the Demo App and Database Credentials

As stated earlier, you'll deploy an existing ASP.NET Core application. This application was built to create a movie list and it uses the Model-View-Controller design pattern to ensure a proper structure and separation of concerns. To create or add a new movie to the list, the user will populate the form fields with the appropriate details and click on the **Create** button to post the details to the controller. The controller at this point will receive a POST HTTP request with the submitted details and persist the data in the database through the model.

You will use Git to pull the source code of this demo application from GitHub and save it in a new directory. You could also download an alternate application here if you will be deploying a different application.

To begin, create a new directory named `movie-app` from the terminal by using the following command:

```
$ sudo mkdir -p /var/www/movie-app
```

This will serve as the root directory for your application. Next, change the folder owner and group in order to allow a non-root user account to work with the project files:

```
$ sudo chown sammy:sammy /var/www/movie-app
```

Replace **sammy** with your sudo non-root username.

```
Output
Cloning into 'movie-app'…
remote: Enumerating objects: 91, done.
```

```
Unpacking objects: 100% (91/91), done.
```

You have successfully cloned the demo application from GitHub, so the next step will be to create a successful connection to the application database. You will do this by editing the `ConnectionStrings` property within the `appsettings.json` file and add the details of the database.

Change directory into the application:

```
$ cd movie-app
```

Now open the file for editing:

```
$ sudo nano appsettings.json
```

Add your database credentials:

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MovieContext": "Server=localhost;User Id=movie-admin;Password=password;Databas
```

ASP.NET Core applications use a .NET standard library named Entity Framework (EF) Core to manage interaction with the database. Entity Framework Core is a lightweight, cross-platform version of the popular Entity Framework data access technology. It is an object- using any

You can now update your database with the tables from the cloned demo application. Run the following command for that purpose:

```
$ dotnet ef database update
```

This will apply an update to the database and create the appropriate schemas.

Now, to build the project and all its dependencies, run the following command:

```
$ dotnet build
```

You will see output similar to:

```
Output
Microsoft (R) Build Engine version 16.1.76+g14b0a930a7 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 95.09 ms for /var/www/movie-app/MvcMovie.csproj.
  MvcMovie -> /var/www/movie-app/bin/Debug/netcoreapp2.2/MvcMovie.dll
  MvcMovie -> /var/www/movie-app/bin/Debug/netcoreapp2.2/MvcMovie.Views.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.91
```

You will see the following:

```
Output
```

```
Restore completed in 89.62 ms for /var/www/movie-app/MvcMovie.csproj.
MvcMovie -> /var/www/movie-app/bin/Debug/netcoreapp2.2/MvcMovie.dll
MvcMovie -> /var/www/movie-app/bin/Debug/netcoreapp2.2/MvcMovie.Views.dll
MvcMovie -> /var/www/movie-app/bin/Debug/netcoreapp2.2/publish/
```

This will pack and compile the application, read through its dependencies, publish the resulting set of files into a folder for deployment, and produce a cross-platform `.dll` file that uses the installed .NET Core runtime to run the application.

By installing dependencies, creating a connection to the database, updating the database with the necessary tables, and publishing it for production, you've completed the setup for this demo application. In the next step you will configure the web server to make the application accessible and secure at your domain.

Step 4 — Configuring the Web Server

By now, having followed the How To Secure Nginx with Let's Encrypt tutorial, you'll have a server block for your domain at `/etc/nginx/sites-available/your_domain` with the `server_name` directive already set appropriately. In this step, you will edit this server block to configure Nginx as a reverse proxy for your application. A reverse proxy is a server that sits in front of web servers and forwards every web browser's request to those web servers. It receives all requests from the network and forwards them to a different web server.

In the case of an ASP.NET Core application, Kestrel is the preferred web server that is included with it by default. It is great for serving dynamic content from an ASP.NET Core application as it provides better request-processing performance and was designed to make ASP.NET as fast as possible. However, Kestrel isn't considered a full-featured web

Open the server block for editing with:

As detailed in the Step 4 of the How To Secure Nginx with Let's Encrypt tutorial, if you selected option 2, Certbot will automatically configure this server block in order to redirect HTTP traffic to HTTPS with just a few modifications.

Continue with the configuration by editing the first two blocks in the file to reflect the following:

/etc/nginx/sites-available/your-domain

```
server {

    server_name your-domain  www.your-domain;

   location / {
     proxy_pass http://localhost:5000;
     proxy_http_version 1.1;
     proxy_set_header Upgrade $http_upgrade;
     proxy_set_header Connection keep-alive;
     proxy_set_header Host $host;
     proxy_cache_bypass $http_upgrade;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
     proxy_set_header X-Forwarded-Proto $scheme;
    }

listen [::]:443 ssl ipv6only=on; # managed by Certbot
listen 443 ssl; # managed by Certbot
ssl_certificate /etc/letsencrypt/live/your-domain/fullchain.pem; # managed by Certb
ssl_certificate_key /etc/letsencrypt/live/your-domain/privkey.pem; # managed by Cer
include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
```

on port `443` and forward every matching request to the built-in Kestrel server at `http://localhost:5000`.

Finally, following the server block you just edited in the file, ensure that the second server

/etc/nginx/sites-available/your-domain

```
…
server {
if ($host = www.your-domain) {
    return 301 https://$host$request_uri;
} # managed by Certbot


if ($host = your-domain) {
    return 301 https://$host$request_uri;
} # managed by Certbot


    listen 80;
    listen [::]:80;

    server_name your-domain  www.your-domain;
return 404; # managed by Certbot
}
```

This server block will redirect all requests to `https://your-domain` and `https://www.your-domain` to a secure HTTPS access.

Next, force Nginx to pick up the changes you've made to the server block by running:

```
$ sudo nginx -s reload
```

With the Nginx configuration successfully completed, the server is fully set up to forward

Move into the `systemd` directory:

```
$ cd /etc/systemd/systems
```

```
$ sudo nano movie.service
```

Add the following content to it:

movie.service

```
[Unit]
Description=Movie app

[Service]
WorkingDirectory=/var/www/movie-app
ExecStart=/usr/bin/dotnet /var/www/movie-app/bin/Debug/netcoreapp2.2/publish/MvcMov
Restart=always
RestartSec=10
SyslogIdentifier=movie
User=sammy
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false

[Install]
WantedBy=multi-user.target
```

The configuration file specifies the location of the project's folder with `WorkingDirectory` and the command to execute at the start of the process in `ExecStart`. In addition, you've used the `RestartSec` directive to specify when to restart the `systemd` service if the .NET runtime service crashes.

Now save the file and enable the new movie service created with:

Then check its status:

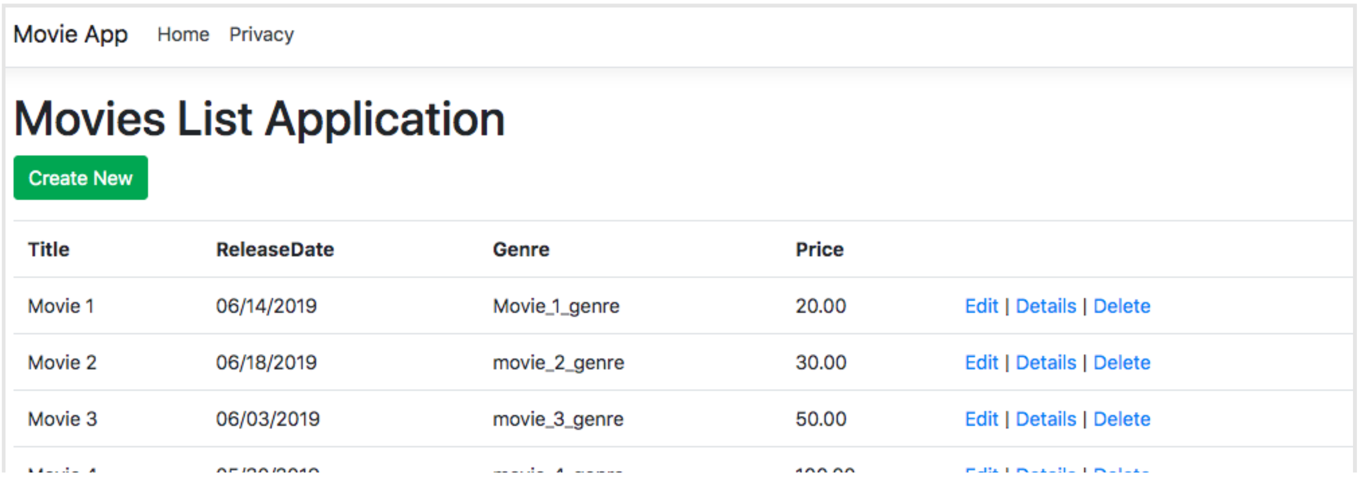
You will see the following output:

```
Output
movie.service - Movie app
   Loaded: loaded (/etc/systemd/system/movie.service; enabled; vendor preset: enabl
   Active: active (running) since Sun 2019-06-23 04:51:28 UTC; 11s ago
 Main PID: 6038 (dotnet)
    Tasks: 16 (limit: 1152)
   CGroup: /system.slice/movie.service
           └─6038 /usr/bin/dotnet /var/www/movie-app/bin/Debug/netcoreapp2.2/publis
```

This output gives you an overview of the current status of the `movie.service` created to keep your app running. It indicates that the service is enabled and currently active.

Navigate to `https://your-domain` from your browser to run and test out the application.

You'll see the home page for the demo application—**Movie List Application**.



Movies List Application

Create New

Title	ReleaseDate	Genre	Price	
Movie 1	06/14/2019	Movie_1_genre	20.00	Edit | Details | Delete
Movie 2	06/18/2019	movie_2_genre	30.00	Edit | Details | Delete
Movie 3	06/03/2019	movie_3_genre	50.00	Edit | Details | Delete

In this tutorial, you deployed an ASP.NET Core application to an Ubuntu server. To persist and manage data, you installed and used MySQL server and used the Nginx web server as a reverse proxy to serve your application.

on using Blazor. It is an event-driven component-based web UI for implementing logic on the client side of an ASP.NET Core application.

If you wish to deploy your own application, you'll need to consider other required procedures to deploy your app. The complete source code for this demo application can be found here on GitHub.

By Oluyemi Olususi

Editor: Kathryn Hancox

Related

TUTORIAL

How To Add the gzip Module to Nginx on

TUTORIAL

How To Add the gzip Module to Nginx on