ASP.NET Core 2.1

Version

2.2 Preview 1

2.1

2.0

1.1

1.0

Host ASP.NET Core on Linux with Nginx

05/22/2018 12 minutes to read Contributors all

In this article

By Sourabh Shirhatti

This guide explains setting up a production-ready ASP.NET Core environment on an Ubuntu 16.04 server. These instructions likely work with newer versions of Ubuntu, but the instructions haven't been tested with newer versions.

For information on other Linux distributions supported by ASP.NET Core, see Prerequisites for .NET Core on Linux.

> **Note**
>
> For Ubuntu 14.04, *supervisord* is recommended as a solution for monitoring the Kestrel process. *systemd* isn't available on Ubuntu 14.04. For Ubuntu 14.04 instructions, see the **previous version of this topic**.

This guide:

- Places an existing ASP.NET Core app behind a reverse proxy server.
- Sets up the reverse proxy server to forward requests to the Kestrel web server.
- Ensures the web app runs on startup as a daemon.
- Configures a process management tool to help restart the web app.

Prerequisites

1. Access to an Ubuntu 16.04 server with a standard user account with sudo privilege.
2. Install the .NET Core runtime on the server.
   a. Visit the .NET Core All Downloads page.
   b. Select the latest non-preview runtime from the list under **Runtime**.
   c. Select and follow the instructions for Ubuntu that match the Ubuntu version of the server.
3. An existing ASP.NET Core app.

Publish and copy over the app

Configure the app for a framework-dependent deployment.

Run dotnet publish from the development environment to package an app into a directory (for example, *bin/Release/<target_framework_moniker>/publish*) that can run on the server:

```
console
dotnet publish --configuration Release
```

The app can also be published as a self-contained deployment if you prefer not to maintain the .NET Core runtime on the server.

Copy the ASP.NET Core app to the server using a tool that integrates into the organization's workflow (for example, SCP, SFTP). It's common to locate web apps under the *var* directory (for example, *var/aspnetcore/hellomvc*).

> **Note**
>
> Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Test the app:

1. From the command line, run the app: `dotnet <app_assembly>.dll`.
2. In a browser, navigate to `http://<serveraddress>:<port>` to verify the app works on Linux locally.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

> **Note**

Either configuration—with or without a reverse proxy server—is a valid and supported hosting configuration for ASP.NET Core 2.0 or later apps. For more information, see **When to use Kestrel with a reverse proxy**.

Use a reverse proxy server

Kestrel is great for serving dynamic content from ASP.NET Core. However, the web serving capabilities aren't as feature rich as servers such as IIS, Apache, or Nginx. A reverse proxy server can offload work such as serving static content, caching requests, compressing requests, and SSL termination from the HTTP server. A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, a single instance of Nginx is used. It runs on the same server, alongside the HTTP server. Based on requirements, a different setup may be chosen.

Because requests are forwarded by reverse proxy, use the Forwarded Headers Middleware from the Microsoft.AspNetCore.HttpOverrides package. The middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

Any component that depends on the scheme, such as authentication, link generation, redirects, and geolocation, must be placed after invoking the Forwarded Headers Middleware. As a general rule, Forwarded Headers Middleware should run before other middleware except diagnostics and error handling middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing.

ASP.NET Core 2.x | ASP.NET Core 1.x

Invoke the UseForwardedHeaders method in `Startup.Configure` before calling UseAuthentication or similar authentication scheme middleware. Configure the middleware to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers:

C# Copy

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

If no ForwardedHeadersOptions are specified to the middleware, the default headers to forward are `None`.

Additional configuration might be required for apps hosted behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Install Nginx

Use `apt-get` to install Nginx. The installer creates a *systemd* init script that runs Nginx as daemon on system startup.

bash Copy

```
sudo -s
nginx=stable # use nginx=development for latest development version
add-apt-repository ppa:nginx/$nginx
apt-get update
apt-get install nginx
```

The Ubuntu Personal Package Archive (PPA) is maintained by volunteers and isn't distributed by nginx.org. For more information, see Nginx: Binary Releases: Official Debian/Ubuntu packages.

> **Note**
>
> If optional Nginx modules are required, building Nginx from source might be required.

Since Nginx was installed for the first time, explicitly start it by running:

bash Copy

```
sudo service nginx start
```

Verify a browser displays the default landing page for Nginx. The landing page is reachable at `http://<server_IP_address>/index.nginx-debian.html`.

Configure Nginx

To configure Nginx as a reverse proxy to forward requests to your ASP.NET Core app, modify */etc/nginx/sites-available/default*. Open it in a text editor, and replace the contents with the following:

nginx Copy

```
server {
    listen        80;
    server_name   example.com *.example.com;
    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;
    }
}
```

When no `server_name` matches, Nginx uses the default server. If no default server is defined, the first server in the configuration file is the default server. As a best practice, add a specific default server which returns a status code of 444 in your configuration file. A default server configuration example is:

nginx Copy

```
server {
    listen   80 default_server;
    # listen [::]:80 default_server deferred;
    return   444;
}
```

With the preceding configuration file and default server, Nginx accepts public traffic on port 80 with host header `example.com` or `*.example.com`. Requests not matching these hosts won't get forwarded to Kestrel. Nginx forwards the matching requests to Kestrel at `http://localhost:5000`. See How nginx processes a request for more information. To change Kestrel's IP/port, see Kestrel: Endpoint configuration.

> ⊗ **Warning**
>
> Failure to specify a proper **server_name directive** exposes your app to security vulnerabilities. Subdomain wildcard binding (for example, `*.example.com`) doesn't pose this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See **rfc7230 section-5.4** for more information.

Once the Nginx configuration is established, run `sudo nginx -t` to verify the syntax of the configuration files. If the configuration file test is successful, force Nginx to pick up the changes by running `sudo nginx -s reload`.

To directly run the app on the server:

1. Navigate to the app's directory.
2. Run the app's executable: `./<app_executable>`.

If a permissions error occurs, change the permissions:

console Copy

```
chmod u+x <app_executable>
```

If the app runs on the server but fails to respond over the Internet, check the server's firewall and confirm that port 80 is open. If using an Azure Ubuntu VM, add a Network Security Group (NSG) rule that enables inbound port 80 traffic. There's no need to enable an outbound port 80 rule, as the outbound traffic is automatically granted when the inbound rule is enabled.

When done testing the app, shut the app down with `Ctrl+C` at the command prompt.

Monitoring the app

The server is setup to forward requests made to `http://<serveraddress>:80` on to the ASP.NET Core app running on Kestrel at `http://127.0.0.1:5000`. However, Nginx isn't set up to manage the Kestrel process. *systemd* can be used to create a service file to start and monitor the underlying web app. *systemd* is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

bash Copy

```
sudo nano /etc/systemd/system/kestrel-hellomvc.service
```

The following is an example service file for the app:

ini Copy

```
[Unit]
Description=Example .NET Web API App running on Ubuntu

[Service]
WorkingDirectory=/var/aspnetcore/hellomvc
ExecStart=/usr/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false

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

If the user *www-data* isn't used by the configuration, the user defined here must be created first and given proper ownership for files.

Linux has a case-sensitive file system. Setting ASPNETCORE_ENVIRONMENT to "Production" results in searching for the configuration file *appsettings.Production.json*, not *appsettings.production.json*.

> **Note**
>
> Some values (for example, SQL connection strings) must be escaped for the configuration providers to read the environment variables. Use the following command to generate a properly escaped value for use in the configuration file:

```
console
systemd-escape "<value-to-escape>"
```

Save the file and enable the service.

```
bash
systemctl enable kestrel-hellomvc.service
```

Start the service and verify that it's running.

```
systemctl start kestrel-hellomvc.service
systemctl status kestrel-hellomvc.service

● kestrel-hellomvc.service - Example .NET Web API App running on Ubuntu
    Loaded: loaded (/etc/systemd/system/kestrel-hellomvc.service; enabled)
    Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
    CGroup: /system.slice/kestrel-hellomvc.service
            └─9021 /usr/local/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
```

With the reverse proxy configured and Kestrel managed through systemd, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. It's also accessible from a remote machine, barring any firewall that might be blocking. Inspecting the response headers, the `Server` header shows the ASP.NET Core app being served by Kestrel.

```
text
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
```

```
Connection: Keep-Alive
Transfer-Encoding: chunked
```

Viewing logs

Since the web app using Kestrel is managed using `systemd`, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by `systemd`. To view the `kestrel-hellomvc.service`-specific items, use the following command:

bash Copy

```
sudo journalctl -fu kestrel-hellomvc.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago` or a combination of these can reduce the amount of entries returned.

bash Copy

```
sudo journalctl -fu kestrel-hellomvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including authentication middleware (for example, cookie middleware) and cross-site request forgery (CSRF) protections. Even if Data Protection APIs aren't called by user code, data protection should be configured to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection to persist and encrypt the key ring, see:

- Key storage providers in ASP.NET Core
- Key encryption At rest in ASP.NET Core

Securing the app

Enable AppArmor

Linux Security Modules (LSM) is a framework that's part of the Linux kernel since Linux 2.6. LSM supports different implementations of security modules. AppArmor is a LSM that implements a Mandatory Access Control system which allows confining the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configuring the firewall

Close off all external ports that are not in use. Uncomplicated firewall (ufw) provides a front end for `iptables` by providing a command line interface for configuring the firewall.

> ⊗ **Warning**
>
> A firewall will prevent access to the whole system if not configured correctly. Failure to specify the correct SSH port will effectively lock you out of the system if you are using SSH to connect to it. The default port is 22. For more information, see the **introduction to ufw** and the **manual**.

Install `ufw` and configure it to allow traffic on any ports needed.

bash Copy

```
sudo apt-get install ufw

sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

sudo ufw enable
```

Securing Nginx

Change the Nginx response name

Edit *src/http/ngx_http_header_filter_module.c*:

c Copy

```
static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

Configure options

Configure the server with additional required modules. Consider using a web app firewall, such as ModSecurity, to harden the app.

Configure SSL

- Configure the server to listen to HTTPS traffic on port `443` by specifying a valid certificate issued by a trusted Certificate Authority (CA).
- Harden the security by employing some of the practices depicted in the following */etc/nginx/nginx.conf* file. Examples include choosing a stronger cipher and redirecting all traffic over HTTP to HTTPS.
- Adding an `HTTP Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS only.
- Don't add the Strict-Transport-Security header or chose an appropriate `max-age` if SSL will be disabled in the future.

Add the */etc/nginx/proxy.conf* configuration file:

nginx Copy

```
proxy_redirect         off;
proxy_set_header         Host            $host;
proxy_set_header         X-Real-IP       $remote_addr;
proxy_set_header         X-Forwarded-For    $proxy_add_x_forwarded_for;
proxy_set_header    X-Forwarded-Proto $scheme;
client_max_body_size     10m;
client_body_buffer_size 128k;
proxy_connect_timeout    90;
proxy_send_timeout    90;
proxy_read_timeout    90;
proxy_buffers          32 4k;
```

Edit the */etc/nginx/nginx.conf* configuration file. The example contains both `http` and `server` sections in one configuration file.

nginx Copy

```
http {
    include    /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens off;

    sendfile on;
    keepalive_timeout 29; # Adjust to the lowest possible value that makes sense for
your use case.
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream hellomvc{
        server localhost:5000;
    }

    server {
```

```
        listen *:80;
        add_header Strict-Transport-Security max-age=15768000;
        return 301 https://$host$request_uri;
    }

    server {
        listen *:443    ssl;
        server_name     example.com;
        ssl_certificate /etc/ssl/certs/testCert.crt;
        ssl_certificate_key /etc/ssl/certs/testCert.key;
        ssl_protocols TLSv1.1 TLSv1.2;
        ssl_prefer_server_ciphers on;
        ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
        ssl_ecdh_curve secp384r1;
        ssl_session_cache shared:SSL:10m;
        ssl_session_tickets off;
        ssl_stapling on; #ensure your cert is capable
        ssl_stapling_verify on; #ensure your cert is capable

        add_header Strict-Transport-Security "max-age=63072000; includeSubdomains;
preload";
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
        location / {
            proxy_pass  http://hellomvc;
            limit_req   zone=one burst=10 nodelay;
        }
    }
}
```

Secure Nginx from clickjacking

Clickjacking is a malicious technique to collect an infected user's clicks. Clickjacking tricks the victim (visitor) into clicking on an infected site. Use X-FRAME-OPTIONS to secure the site.

Edit the *nginx.conf* file:

bash Copy

```
sudo nano /etc/nginx/nginx.conf
```

Add the line `add_header X-Frame-Options "SAMEORIGIN";` and save the file, then restart Nginx.

MIME-type sniffing

This header prevents most browsers from MIME-sniffing a response away from the declared content type, as the header instructs the browser not to override the response content type. With the `nosniff` option, if the server says the content is "text/html", the browser renders it as "text/html".

Edit the *nginx.conf* file:

```
bash
sudo nano /etc/nginx/nginx.conf
```

Add the line `add_header X-Content-Type-Options "nosniff";` and save the file, then restart Nginx.

Additional resources

- Prerequisites for .NET Core on Linux
- Nginx: Binary Releases: Official Debian/Ubuntu packages
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX: Using the Forwarded header