.NET Core 2.1 Global Tools

Getting started with creating a .NET Core global tool package. Also, a peek under the hood.

By Nate McMaster | May 12, 2018

dotnet nuget

.NET Core 2.1 RC1 was released this week. This is the first supported version of the .NET Core CLI which includes a feature called ".NET Core Global Tools". This feature provides a simple way to create and share cross-platform console tools. In this post, I'll go over some of the basics, and then walk though what is going on under the hood. You will need to download .NET Core 2.1 to use this to try this on your own.

Tip: For a real-world example of a global tool, see https://github.com/natemcmaster/dotnet-serve/.

Basic design

A .NET Core global tool is a special NuGet package that contains a console application. When installing a tool, .NET Core CLI will download and make your console tool available as a new command.

Users install tools by executing "dotnet tool install":

```
> dotnet tool install -g awesome-tool
```

Once installed, the command "awesome-tool" can now be used.

```
> awesome-tool
```

Creating your own package

To simplify getting started, I've created a project templates.

Install the templates package

```
dotnet new --install McMaster.DotNet.GlobalTool.Templates
```

Create a new project

```
dotnet new global-tool --command-name awesome-tool
```

Once you have this project, you can create your tool package like this:

```
dotnet pack --output ./
```

This creates a file named `awesome-tool.1.0.0.nupkg` You can install your package like this:

```
dotnet tool install -g awesome-tool --add-source ./
```

When you are ready to share with world, upload the package to https://nuget.org.

> *Note: in 2.1 RC1, use `--source-feed` instead of `--add-source`. This was renamed in https://github.com/dotnet/cli/pull/9164*

Additional commands

`dotnet tool` has other commands you can invoke. For example,

```
dotnet tool list -g
dotnet tool uninstall -g awesome-tool
dotnet tool update -g awesome-tool
```

Under the hood

The NuGet package that contains the tool is produced by running `dotnet publish` and putting everything in publish output into the NuGet package, with a few manifest files.

When `dotnet tool install --global` executes, it...

1. uses `dotnet restore` with special parameters to fetch the package.
2. extracts the package into `$HOME/.dotnet/.store/(package id)/(version)`
3. Generates an executable in `$HOME/.dotnet/tools`

The executable generated is a small console app (written in C++) which automatically knows how to find your .NET Core .dll file and launch it.

When `dotnet tool install --tool-path $installDir` executes, it does the same thing, but installs into `$installDir`, not `$HOME/.dotnet/tools`

Package authoring and SDK

Authoring a global tool requires the .NET Core SDK. This SDK provides a few simple settings to control the naming and contents of a .NET Core global tool package.

The required minimum project for a global tool looks like this.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <PackAsTool>true</PackAsTool>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Additional properties can be set to control the generated tool.

- `ToolCommandName` - the command name users will invoke. It defaults to the name of the .dll file produced (i.e. assembly name)
  - This does **not** need to start with `dotnet-`. It can be anything without spaces.
  - If it begins with `dotnet-`, it can be used as a subcommand on `dotnet`, provided that command doesn't already exist. Example: `dotnet-serve` can be invoked as either `dotnet serve` or `dotnet-serve`.
- `PackageId` - the ID of the NuGet package (defaults to the .csproj file name)
  - The package ID can be different than the command name and assembly name
- `PackageVersion` - the version of the NuGet package (defaults to 1.0.0)
- `AssemblyName` - can be used to change the file name of the .dll file

Example of using these properties:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <PackAsTool>true</PackAsTool>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <ToolCommandName>pineapple</ToolCommandName>
    <PackageId>dole-cli</PackageId>
    <PackageVersion>1.0.0-alpha-$(BuildNumber)</PackageVersion>
    <AssemblyName>Dole.Cli</AssemblyName>
  </PropertyGroup>
</Project>
```

Deep-dive: package requirements

There are some very specific requirements for global tools. The SDK takes care of most of these for you when you specify PackAsTool=true.

Publish output into pack

As mentioned above, the tools package must contain all your apps dependencies. This can be collected into one place by using `dotnet publish`. By default, `dotnet pack` only contains the output of `dotnet build`. This output does not normally contain third-party assemblies, static files, and the `DotnetToolSettings.xml` file, which is why you need to specify `PackAsTool`. This instructs the SDK to gather all publish output, not just assembly you compile.

DotnetToolSettings.xml

This file is generated by the SDK when you set `PackAsTool` to `true`, and normally, you do not need to create it yourself. The file must exist in the package. If not, dotnet will fail to install your tool with

The settings file in the tool's NuGet package is invalid: Settings file 'DotnetToolSettings.xml' was not found in the package.

The schema for this file looks like this:

```
<DotNetCliTool Version="1">
  <Commands>
    <Command Name="$name" EntryPoint="$file" Runner="$runner" />
  </Commands>
</DotNetCliTool>
```

- `$name` is the same value as `ToolCommandName` in your .csproj. It is the command users will type on the command line to invoke your tool
- `$file` is the main file name for your .NET Core console app, such as `pineapple.dll`
- `$runner` is the name of the command used to invoke your file. As for 2.1, the only allowed value is `dotnet`.

When you install, `dotnet` creates an "alias" (sort of) which maps the command `$name` to invoke `$runner $file`. It would sort of like using bash to set `alias pineapple=dotnet pineapple.dll`.

Currently, there are many restrictions on how this file can be used.

- The file must exist in the NuGet package under `tools/$targetframework/any/`. Example: `tools/netcoreapp2.1/any/DotnetToolSettings.xml`.
- You may only specify one `DotnetToolSettings.xml` file per package.
- You may only specify one `<Command>` per `DotnetToolSettings.xml` file.
- The only allowed value for `Runner` is `"dotnet"`.
- The value for `EntryPoint` must be a `.dll` file that sits next to `DotnetToolSettings.xml` in the package.

Error NU1212 and package type

Installation may fail with this error

```
error NU1212: Invalid project-package combination for awesome-tool 1.0.0. DotnetToolRef
```

This error message is not very clear (see https://github.com/NuGet/Home/issues/6630 for improvement). What this means is that dotnet-tool-install is currently restricted to only installing a .NET Core package that has specific metadata. That metadata can be defined in your nuspec file and must be set as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2012/06/nuspec.xsd">
  <metadata>
    <!-- This piece is required -->
    <packageTypes>
      <packageType name="DotnetTool" />
    </packageTypes>
    <!-- ... -->
  </metadata>
</package>
```

Dependencies

You must redistribute any of your dependencies in your tools package. Dependencies defined in the `<dependencies>` metadata of your NuGet package are not honored by dotnet-tool-install.

Common errors

Here are some common errors encountered when using global tools.

PATH and command not after installing

```
> dotnet tool install -g awesome-tool

> awesome-tool
awesome-tool: command not found
awesome-tool : The term 'awesome-tool' is not recognized as the name of a cmdlet, functi
```

Global tools are actually “user global”, and are installed to `%USERPROFILE%\.dotnet\tools` (Windows) or `$HOME/.dotnet/tools` (macOS/Linux). The .NET Core CLI tool makes a best effort to help you ensure this is in your PATH environment variable. However, this can easily be broken. For instance:

- if you have set the `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` environment variable to speed up intial runs of .NET Core, then your PATH may not be updated on first use
- **macOS**: if you install the CLI via `.tar.gz` and not `.pkg`, you’ll be missing the

`/etc/paths.d/dotnet-cli-tool` file that configures PATH.

- **Linux:** you will need to edit your shell environment file. e.g. `~/.bash_profile` or `~/.zshrc`

Workarounds

(May require restarting your shell.)

Windows:

```
setx PATH "$env:PATH;$env:USERPROFILE/.dotnet/tools"
```

Linux/macOS:

```
echo "export PATH=\"\$PATH:\$HOME/.dotnet/tools\"" >> ~/.bash_profile
```

Tools are user-specific, not machine global

The .NET Core CLI installs global tools to `$HOME/.dotnet/tools` (Linux/macOS) or `%USERPROFILE%\.dotnet\tools` (Windows). This means you cannot install a global tool for the entire machine using `dotnet tool install --global`. Installed tools are only available to the user who installed them.

Installing the .NET Core CLI into a non-default location

If you download the .NET Core CLI as a .zip/.tar.gz and extract it to a non default location, then you may encounter an error after installing and launching a tool:

- **Windows**: `A fatal error occurred, the required library hostfxr.dll could not be found`
- **Linux**: `A fatal error occurred, the required library libhostfxr.so could not be found`
- **macOS**: `A fatal error occurred, the required library libhostfxr.dylib could not be found`

The error will also contain additional details such as:

```
If this is a self-contained application, that library should exist in [some path here].
If this is a framework-dependent application, install the runtime in the default locatio
```

The reason this happens is that the generated shim created by `dotnet tool install` only searchs for .NET Core in the default install locations. You can override the default location by setting the DOTNET_ROOT environment variable.

```
set DOTNET_ROOT=C:\Users\nate\dotnet
export DOTNET_ROOT=/Users/nate/Downloads/dotnet
```

See https://github.com/dotnet/cli/issues/9114 for more details.

Wrapping up

This is an awesome feature. Super happy the .NET Core CLI team created it. Can't wait to see what people make.