Building Rationally: Easy CI

In this lesson, you will learn how Docker helps in building software and eases integration.

Note that I wrote *"build"* in the preceding chapter, while most of what we did in our *Dockerfile* files was pack our software, not build it.

But I really meant *build*. Docker is not only a technology to pack your software, but it can build it too. When you build an image, you are actually running software inside a container, the *RUN* commands in the *Dockerfile* file. Since those *RUN* commands run in an image, the image describes the dependencies needed to build the software.

Remember those hard times you had as a developer? You cloned some code only to realize that building it took you a day since you needed to install many build dependencies and SDK tools. Once you decide to build your software inside images, all you need is Docker. Whether on a development machine or CI/CD server, Docker will be helpful. This is a major benefit for the projects I've been working on, and I have no doubt that you're going to benefit from it as well.

This can even be taken a step further; you can use the same Docker machines to build and run your software. Instead of having a build server and a test server, you can use the test server to build. Going a step further with orchestration, you can use the same Kubernetes cluster to host your deployments *and* build your code - but that's another story.

Let's take an ASP.NET Core application as an example. ASP.NET Core requires us to restore the NuGet packages referenced by the *.csproj* files, then build the source code (C# files) into DLLs and pack in any necessary dependencies. This can be done with the following *Dockerfile* definition:

Dockerfile

```
FROM microsoft/dotnet:2.2-sdk AS builder
WORKDIR /app

COPY . .
RUN dotnet restore
RUN dotnet publish --output /out/ --configuration Release

EXPOSE 80
ENTRYPOINT ["dotnet", "aspnet-core.dll"]
```

The *FROM* instruction makes sure we have the tools needed to build (SDK) then calls *dotnet restore* and *dotnetpublish* to get our code ready for deployment inside the image. This allows us to run *docker build* on the CI server or a developer machine and get an image that contains built files. When the container starts on a server *(docker run)*, the *ENTRYPOINT* instruction makes sure that our code has started inside the container.

Since the *Dockerfile* definition is archived with our source code, we are sure that we'll keep the build instructions synchronized with our code, and different branches may even have different *Dockerfile* definitions.

However, there's a problem with the resulting image. Can you guess what it is? Let's find out in the next lesson.