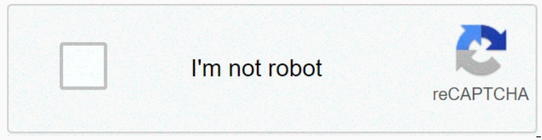

Continue

Msbuild allow nuget to missing packages

By default, the NuGet.Config file instructs NuGet to bypass adding package binary files to the source control. # Ignore NuGet Packages *.nupkg # Ignore the packages folder **/packages/* # except build/, used as MSBuild meta. !**/packages/build/ # Uncomment if necessary however it will generally regenerate when needed #!**/packages/repository.config NuGet offers three approaches to using package restore. When building a solution from the command line, a renewal of the command package is required; was introduced in early versions of NuGet, but improved in NuGet 2.7. The MSBuild-integrated package renewal approach is the original implementation of package restore and while it still works in many scenarios, it does not cover the full range of scenarios solved by the other two approaches. Automatic Package Restore is the NuGet team's recommended approach to package renewal within Visual Studio, and is presented in NuGet 2.7. nuget restore TheSolutionFilname.sln Old way, right-click on your solution in VS and choose Enable Package Restore. This causes VS to modify csproj files, and create a .nuget folder that contains nuget.exe and some other files. NuGet changes the project files in the solution to refer to the NuGet.targets file so that it can participate in the construction process. the presence of the NuGet.targets file determines whether NuGet will continue to use the MSBuild-Integrated approach. Each project file contains a fragment as <RestorePackages>true</RestorePackages> ... <Import project=$(SolutionDir)\.nugetuget.targets></Import> ... <Target name=EnsureNuGetPackageBuildImports beforetargets=PrepareForBuild> <PropertyGroup> <ErrorText>This project points to the NuGet package(s) missing from this computer. Enable the NuGet Package Restore to download them. For more information, see . The missing file {0}.</ErrorText> </PropertyGroup>Automatic renewal of Nuget has changed to Nuget 2.7+. <Error condition=! Exists('$(SolutionDir)\.nuget\NuGet.targets') text=$([System.String]::Format('$(ErrorText)', '$(SolutionDir)\.nuget\NuGet.targets'))></Error> </Target> Do not mix old and new methods for automatic renewal of packages. Edit each project file (e.g. csproj, .vbproj) in the solution and remove all references to the NuGet.targets file. Open the project(file) file in the editor of your choice and </PropertyGroup> <PropertyGroup Condition='$(OS)' != condition=></PropertyGroup Condition='$(OS)' != > </Project> </Project> <NuGetExe>nuget</NuGetExe> <Target name=RestorePackages beforetargets=Build dependsontargets=GetNuGet> <Exec command="$(NuGetExe)" Restore "$(SolutionPath)" condition='$(SolutionPath)' != "></Exec> <Exec command="$(NuGetExe)" Restore "%(Solution.Identity)" condition='%(Solution.Identity)' != "></Exec> </Target> <PropertyGroup condition='$(OS)' == 'Windows_NT'> <CodeTaskAssembly condition='$(MSBuildAssemblyVersion)' == ">$(MSBuildToolsPath)\Microsoft.Build.Tasks.v4.0.dll</CodeTaskAssembly> <!-- In VS2013, the assembly contains the VS version. --> <CodeTaskAssembly condition='$(MSBuildAssemblyVersion)' == '12.0'>$(MSBuildToolsPath)\Microsoft.Build.Tasks.v12.0.dll</CodeTaskAssembly> <!-- In VS2015+, the assembly was renamed, hopefully this will be the last condition! --> <CodeTaskAssembly condition='$(MSBuildAssemblyVersion)' != '' and '$(MSBuildAssemblyVersion)' >= '14.0'>$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll Ovdje</CodeTaskAssembly> </PropertyGroup> <UsingTask taskname=DownloadNuGet taskfactory=CodeTaskFactory assemblyfile=$(CodeTaskAssembly) condition='$(OS)' == 'Windows_NT'> <ParameterGroup> <TargetPath parametertype=System.String required=true></TargetPath> </ParameterGroup> <Task> <Reference include=System.Core></Reference> <Using namespace=System></Using> <Using namespace=System.IO></Using> <Using namespace=System.Net></Using> <Using namespace=Microsoft.Build.Framework></Using> <Using namespace=Microsoft.Build.Utilities></Using> <Code type=Fragment language=cs> <![CDATA[try { TargetPath = Path.GetFullPath(TargetPath); if (! Directory.Exists(Path.GetDirectoryName(TargetPath))) Directory.CreateDirectory(Path.GetDirectoryName(TargetPath)); Log.LogMessage(Downloading latest version of NuGet.exe...); WebClient webClient = new WebClient(); webClient.DownloadFile(TargetPath); return true; } catch (Exception ex) { Log.LogErrorFromException(ex); return false; }]] > </Code> </Task> </UsingTask> <Target name=GetNuGet condition='$(OS)' == 'Windows_NT' And ! Exists('$(NuGetExe)')> <MakeDir directories=$(NuGetPath) condition=! Exists($(NuGetPath))></MakeDir> <DownloadNuGet targetpath=$(CachedNuGet) condition=! Exists($(NuGetExe)) And ! Exists($(CachedNuGet))></DownloadNuGet> <Copy sourcefiles=$(CachedNuGet) destinationfolder=$(NuGetPath) condition=! Exists($(NuGetExe))></Copy> </Target> vidimo, to samo osigurava, da je naredba za vraćanje grumena izvršena jednom na početku procesa izgradnje može biti odsutna ili samo na pogrešnom relativnom putu: Kao We all know NuGet is a package manager that is widely used in the Microsoft stack. To promote clean implementations and reduce repositor size, it is always advisable to pack shared libraries divided into multiple applications. In this post, I will share some of my findings with the NuGet Package Manager on how we can NuGet rebuild with devOps CICD pipeline. Before we start let me set up a backgound first. If you use VSTS for your CICD pipeline, it already has NuGet to restore to build the task. So configuring your NuGet restore task in your construction pipeline would be really easy. But in my case we had to go to the AWS CICD stack, which consists of AWS Lamda, AWS code pipeline, code implementation, S3 buckets, etc. If you are interested to learn more about CICD pipelines in the AWS stack visit the link below:For .Net applications generally build pipeline will consist of powershell scripts to call msbuild to build applications and post wherever needed. Typically, your build boxes for .Net applications (e.g. ASP.NET web application, ASP.Net MVC applications) would require a .Net box and msbuild.exe installed in them. Now before building your app we would require downloading/returning all the necessary packages that are needed to build this application, right? Here are some options that are there to create that package to restore the step in building the pipeline: Package Manager UI (Visual Studio on Windows): Now allows you to keep dotnet core side and talk about .NET applications. The package manager comes as part of Visual Studios, so when you build a project using a studio visual editor, it has the ability to download packages automatically, or you can manually use the package manager's AI and return the missing packages. But that wouldn't be very helpful in our CICD pipeline, would it? NuGet CLI: Use the Restore Nums command, which returns packages specified in a file project or in packages.config. You can also specify a solution file. MSBuild: Use msbuild -t:restore command, which returns package packages specified in the file project (PackageReference only). Available only in NuGet 4.x+ and MSBuild 15.1+, which are included in Visual Studio 2017. Restoring nuggets and restoring dotnets use this command for applicable projects. More in the link below:Now that we have seen the three options let us talk in a little detail. The AI package manager will not serve our purpose in automated environments such as pipeline construction, keeping this aside allows us to focus on NuGet CLI and MSBuild.MSBuild: As previously stated, msbuild 15.1+ has a t:restore option that is a target:restore option that can restore NuGet packages to your project, but the only limitation with the target restore is that it only works for packageReference format in the project. For more information about packagereference format, visit the link below:Since most projects follow the package.config format for the NuGet package reference, it will be an additional job to migrate the package.config to PackageReference format. Thus, the NuGet CLI client tool will fit perfectly for the construction of pipelines for the NuGet package restore from .Net applications. NuGet CLI: NuGet Command Line Interface (CLI), nuget.exe, provides full scope of NuGet functionality for installing, creating, publishing and managing packages any changes in the Files. To use this tool in our pipeline for construction, we would have to install it in our building box or keep this as part of our source code. Installing CLI in the construction frame would be the preferred solution, as keeping this one in the source would take up additional space resulting in longer cloning times. To know more about NuGet CLI, visit the link below:I hope this post will be useful for those who build the DevOps CICD pipeline or build scripts for .Net applications. Please let me know in the comments whether I have missed any other information here or if there are other better ways to achieve it. Until then, happy :) :)