# Chapter 6: Building Code

In Chapter 5, you learned how to properly organize your Git repository in preparation of DevOps automation. In this chapter, you will learn how to build the code. You will learn the difference in a private build and an integration build, often called a continuous integration build or CI build. And you will learn how to configure your CI build in Azure DevOps Services. If you are following along in the code, make sure you have cloned the sample application.

https://clearmeasurelabs@dev.azure.com/clearmeasurelabs/

Onion-DevOps-Architecture/\_git/

Onion-DevOps-Architecture

## Structure of a build

In 2007, Paul Duvall, Steve Matyas, and Andrew Glover published a book called “Continuous Integration: Improving Software Quality and Reducing Risk”[[1]](#footnote-1). At the time, continuous integration was a new topic, and the industry was conducting a far-reaching conversation. This book documented the proven structure for the practice of continuous integration. In it, the two types of builds were clearly defined:

* Private build
* Integration build

The private build only runs on a single developer workstation, and it is a tool to know that immediate changes did not destabilize the application. The integration build runs on a shared server and belongs to the team. It builds code from many developers. With the rise in popularity of branching models, the integration build has been adapted to run on feature branches as well as the master branch. Before we move on to how to implement our builds, let’s review the structure and flow of a build process.

### Flow of a build on a feature branch

Before we discuss the steps of a private build or a CI build, let’s look at it from a high level. When you start work on a user story or software change, regardless of branching strategy, you will create a branch. Remember in Chapter 4, you learned that even those using Trunk-based Development still use short-lived branches. The figure below shows the flow of build activities that happen when you are working on a feature branch.



Figure: The build process for code on a feature branch flows across three environments.

When you change code, you will run your private build at every stopping point. This keeps you safe. You will learn right away if you accidentally broke something. Because you at working in Git, a decentralized Version Control System, you’ll make many, short commits. This enables you to undo changes very easily. Based on your judgement, you’ll run the private build locally. In our application, it is a PowerShell script and is described in more detail later in this chapter. When you decide to push changes to your team’s Git server, the CI build will detect those changes and run the integration build process on the team’s build server. Upon success, the build will archive the built artifacts, most likely in Azure Artifacts, a Nuget repository. Then an automated deployment script will trigger and deploy those built artifacts to an environment dedicated to the continuous integration process. The best name for this environment is the “TDD Environment”. The purpose of this environment is to validate that 1) the new version of the software can be deployed and 2) the new version of the software still passes all its acceptance tests. This does require that you have full-system acceptance tests in your code base. If you don’t, they are easy to start developing. After the acceptance tests succeed and you determine your changes are complete, you, as the developer, will create a pull request so that your team knows that you believe the work on your branch is complete and that the code is ready to be inspected for inclusion in the master branch.

### Flow of a build on the master branch

Once a pull request has been approved, your branch is automatically merged into master. This is true whether you are using GitHub or Azure Repos. The CI build, which is monitoring for changes, will initiate. Upon success, the build artifacts will be stored in Azure Artifacts as Nuget packages. Then the build will be deployed to the TDD environment for validation of deployability and for the running of the automated full-system acceptance tests. Once these acceptance tests complete successfully, the build is considered a valid release candidate. That is, it is a numbered candidate for potential release and can be validated further in manual testing environments (or even additional automated testing environments) and deployed along the pipeline toward production. The following figure shows the lifecycle of a master branch build.

The deployable package for a software build can be as simple as a zip file, but in .Net, the Nuget package is the standard, and these are meant to be archived in Azure Artifacts.



Figure: The build process for changes on master end with a new release candidate.

### Steps of a build

Before we walk through how to configure a build on your own workstation and in Azure Pipelines, let’s review the steps a private build and a CI build must have.



Figure: The private and CI build have many steps in common.

The private build runs on a developer workstation. The CI build runs on shared team build infrastructure, whether a full server or in Azure Pipelines. Test-driven development[[2]](#footnote-2) (TDD) introduced the validation concept of Arrange, Act, Assert. Here is the flow.

1. Arrange: In any validation, whether an automated test, a manual test, a static analysis run, or a CI build, the validation process is responsible for setting up an environment in which it can run.
2. Act: In this step, you execute a process, run some code, kick off a procedure, etc.
3. Assert: Finally, you see how things went. You check to make sure that what did happen was in line with what you expected to happen. If what happened met expectations, your validation has succeeded. If it didn’t meet expectations, your validation has failed.

Just like in TDD, a build process is a formal validation. You will need to add steps to your build script to set up the environment for the build to run (Arrange), run the transition from source files to executable form (Act), and check as many things as you can (Assert). In the figure above, you can see the types of activities that are in both our private build and our CI build. Let’s go through them one by one.

* Start: The private build will be triggered on demand by a developer. The CI build will be triggered by a watcher on the Git repository – when a new commit occurs.
* Clean: Any temporary directories or files are deleted, and any remnants of previous builds are expunged.
* Version: The build number is pushed into any areas of input needed for the resulting executable software to be stamped with the version number of the build. It’s common for a private build to have a hardcoded version such as 0.0.0 or 9.9.9 so that anyone observing can immediately tell that a build is from a private build. In Azure Pipelines, the build number will come in from an environment variable, and the build script should push this number into relevant places, such as an AssemblyInfo.cs file for .Net Framework, or the dotnet.exe command line for .Net Core. If this step is omitted, resulting .Net assemblies will not be properly labeled with the build number.
* Migrate Database: This step represents anything environmental that the application needs in order to function. Most applications store data, so a database needs to be created and migrated to the current schema in preparation for the subsequent build steps. In this book, we show examples using a SQL Server relational database schema.
* Compile: This step transforms source files into assemblies, and performs any encoding, transpiling[[3]](#footnote-3), minification, etc. to turn source code into a form suitable for execution in the intended runtime environment.
* Unit Tests: This is the first step that falls into the Assert category. Now that we have a form of the software that can be validated, presuming the compile step succeeded, we start with the fastest type of validations. Unit tests execute classes and methods that do not call out of process. In .Net, this is the AppDomain, which is the boundary for a space of memory. Therefore, unit tests are blazing fast.
* Integration Tests: These tests ensure that various components of the application can integrate with each other. The most common is that our data access code can integrate with the SQL Server database schema. These tests execute code that traverses across processes (.Net AppDomain, through the networking stack, to the SQL Server process) in order to validate functionality. These tests are important, but they are orders of magnitude slower than unit tests. As an application grows, expect about a 10:1 ratio of unit tests to integration tests.
* Private build success: After these steps, a private build is done. Nothing further is necessary to run on a developer workstation.
* Static code analysis: Whether it be the FxCop family of analyzers, products like Ndepend or SonarQube, or JavaScript linters, a CI build should include static code analysis in its list of validations. They are easy to run and find bugs that automated tests will not. Capers Jones includes them in the top 3 defect detection methods from his research[[4]](#footnote-4).
* Publish test results: At this point, the CI build has succeeded and needs to output the build artifacts. Each application type has a process that outputs the artifacts in a way that is suitable for packaging, which is the next step.
* Package: In .Net, this is the act of taking each deployable application component and compressing it into a named and versioned Nuget package. For example, UI (ASP.NET website), Database (SQL Server schema migration assets), BatchJob (Windows service, Azure Function, etc.), Acceptance Tests (deployable tests to be run in further down the DevOps pipeline). These Nuget packages are to be pushed to Azure Artifacts. While it is possible to use zip files, Nuget is the standard package format for .Net.
* Publish: pushing the packaged Nuget files to Azure Artifacts so they are available through the a Nuget feed.
* CI build success: The continuous integration build has now completed and can report success.

Your implementation of a private build and a CI build can vary from the examples shown in this book but take care to include each of the steps above in a fashion that is suitable for your application. Now that you know the structure of the builds, let’s cover how to configure and run them in a .Net environment.

## Using builds with .Net Core and Azure Pipelines

Azure Pipelines is gaining wide adoption because of the compatibility and ease with which an automated continuous delivery pipeline can be set up with a software application residing anywhere. Whether GitHub or Azure Repos, or your own Git repository, Azure Pipelines can provide the build and deploy pipeline. There are four stages to continuous delivery, as described by the 2010 book, “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”[[5]](#footnote-5). These stages are:

* Commit
* Automated acceptance tests
* Manual validations
* Release

The commit stage includes the private build and continuous integration build. The automated acceptance test stage includes your TDD environment with the test suites that represent acceptance tests. The UAT environment, or whatever name you choose, represents the deployed environment suitable for manual validations. Then, the final release stage goes to production where your marketplace provides feedback on the value you created for it. Let’s look at the configuration of the private build and of Azure Pipelines and see how to enable the Commit stage of continuous delivery.

### Enabling continuous delivery’s Commit stage

Before you configure Azure Pipelines, you must have your private build. Attempting to create a CI build without this foundation is a recipe for lost time and later rework. In the source code that accompanies this book, you will find a PowerShell build script named “./build.ps1”. The full listing for this file is at the end of this chapter. Feel free to use it as a build script for your own .Net Core applications. It contains all the necessary steps narrated above and will serve as a good jump start for your CI build. This build scripts contains steps to restore, compile, create a local database, and run tests. The first time you clone the repository, you’ll see quite a bit of NuGet restore activity that you won’t see on subsequent builds because these packages are cached. The following figure shows the dotnet.exe restore output that you’ll only see the first time after clicking click\_to\_build.bat.

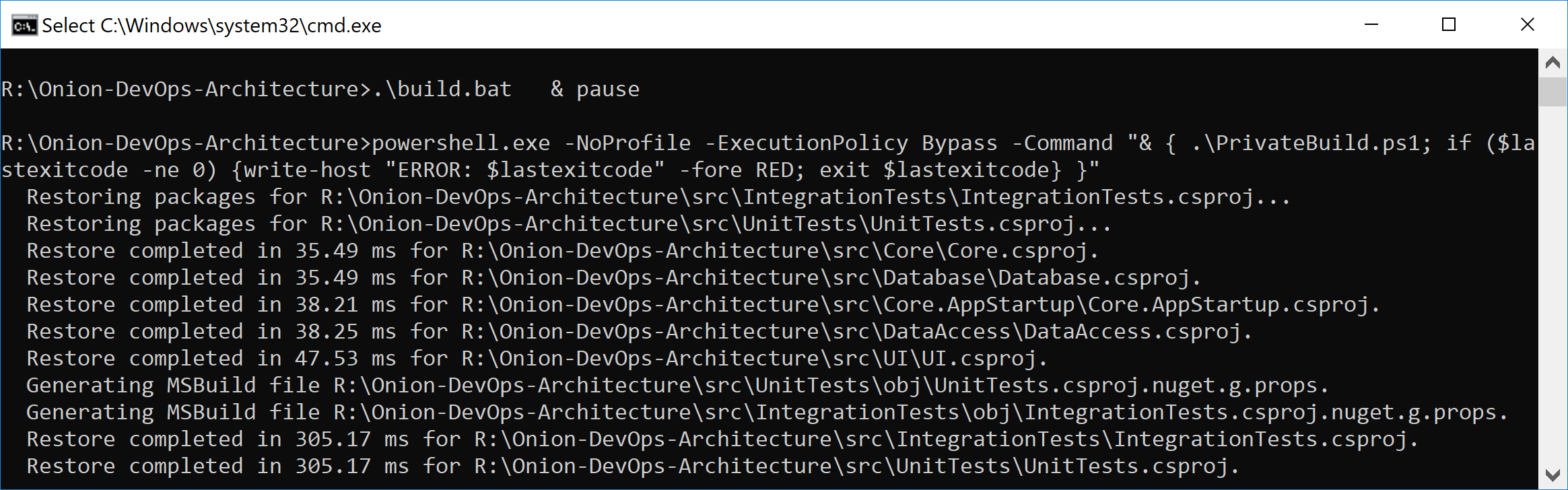


Figure: The first time the private build runs, you’ll see more output than normal from the Restore step.

Click\_to\_build.bat is a simple helper file that makes running a private build easy and convenient by adding a “& pause” so that the command window remains open when invoked by the keyboard or mouse from Windows Explorer. In the normal course of development, you’ll run the private build repeatedly to make sure that every change you’ve made is a solid, stable step forward. You’ll be using a local SQL Server instance, and the build script will destroy and recreate your local database every time you run the script. Unit tests will run against your code. Component-level integration tests will ensure that the database schema and ORM configuration work in unison to persist and hydrate objects in your domain model. The following figure shows the full build script executive with “quiet” verbosity level enabled.

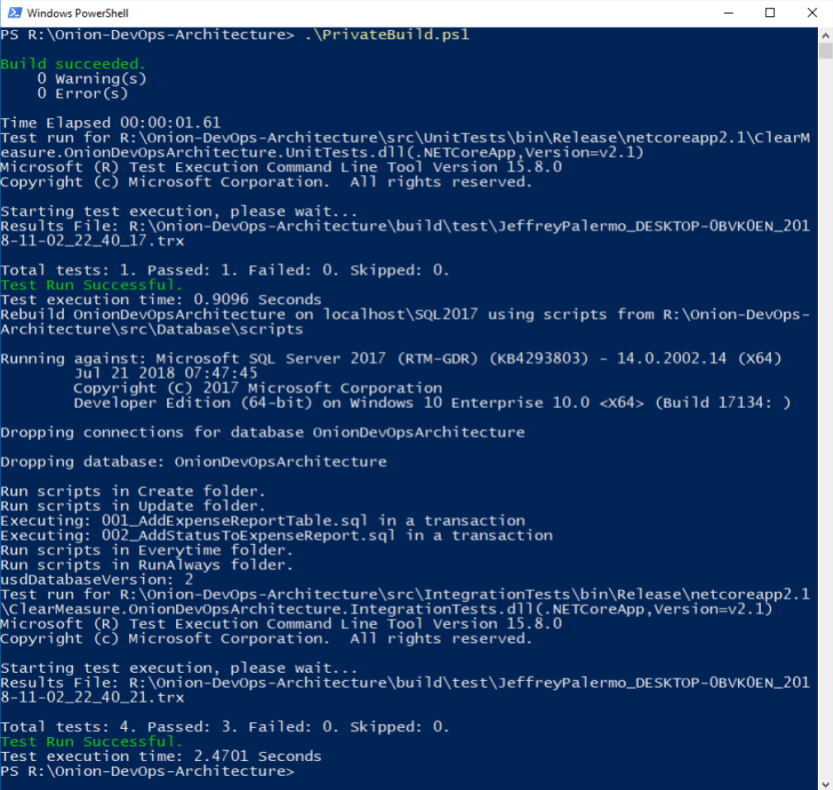


Figure: The output from the private build can fit on one screen and run in less than one minute.

This is a simple private build script, but it scales with you no matter how much code you add to the solution and how many tests you add to these test suites. In fact, this build script doesn’t have to change even as you add table after table to your SQL Server database. This build script pattern has been tested thoroughly over the last 13 years across multiple teams, hundreds of clients, and a build server journey from CruiseControl.NET to Jenkins to Bamboo to TeamCity to VSTS to Azure Pipelines. Although parts and bits might change a little, use this build script to model your own. The structure is proven.

Now that you have your foundational build script, you’re ready to create your Azure Pipeline CI build. As an overview, the following figure shows the steps you use, including pushing your release candidate packages to Azure Artifacts.

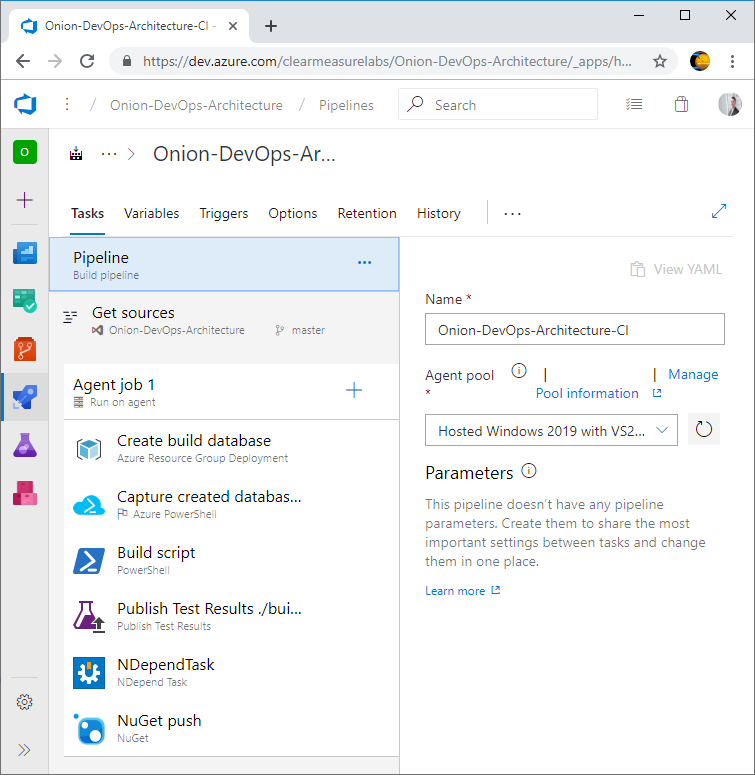


Figure: Azure Pipelines build configuration is quite simple when you start with the foundation of a private build script.

Many of the defaults are suitable for CI builds and don’t have to be customized. Let’s go through the parts that are important. First, you’ll choose your agent pool. I’ve chosen hosted agent for Visual Studio 2019. For the purposes of illustration, I’m using the build designer rather than the YAML option. All the builds and release definitions in Azure Pipelines are being converted to YAML. At the time of this writing, the YAML tooling, editor, and marketplace integration were not yet deployed. Because of this, the designer provides the full editing experience. Expect the YAML experience to enhanced quickly. When it is fully complete, you’ll be able to save your CI build configuration as a YAML file in your Git repository right next to your application. You will want to do this because any logic not versioned with your code could break your pipeline since it is inherently not compatible with branching given that only one version of the build configuration exists.

To continue down the CI build configuration, you need to set up the environment for the execution of the PowerShell build script that contains the shared steps with our private build. This means that I need a SQL Server database. Given that the hosted build agents don’t have a SQL Server installed on them, I’ll need to go elsewhere for it. You can use an ARM script to provision a database in your Azure subscription so that your integration tests have the infrastructure with which to test the data access layer. The ARM scripts for this are part of the sample application. After the creation of a database that can be used by the integration tests, you want to ensure that your compilation steps handle the versioning properly. After all, the purpose of this build is to create a release candidate. The candidate for release must be versioned and packaged properly and then run through a gauntlet of validations before you would ever trust it to run in production. As you call your PowerShell build script, you call the command with the following arguments

./build.ps1 ; CIBuild

Even though there is only one explicit parameter in the above command, all the build variables are available to any script as environment variables. The next figure shows the variables that are configured for this build.

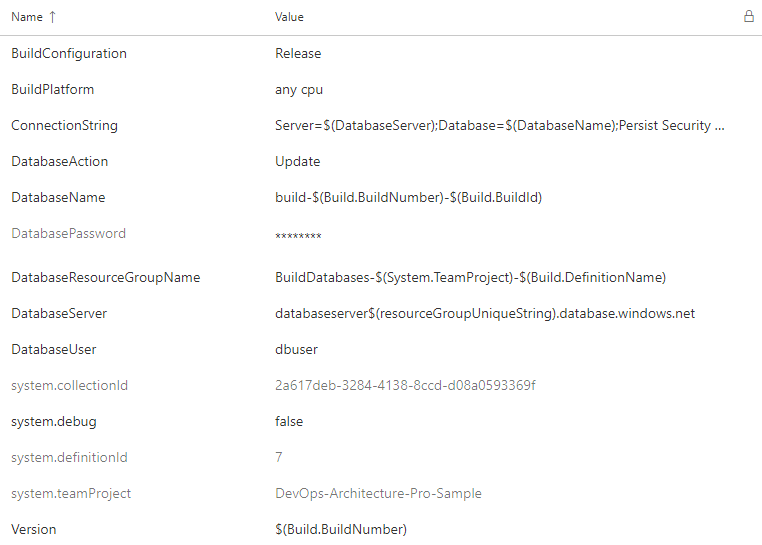


Figure: The build variables are available to the build steps as environment variables.

As variables are defined, make use of other variables in order to build up the appropriate values. You will find that once you create a few CI build configurations and variable sets, the patterns are very portable from one application to the next. Make sure to vary values so that multiple builds can run in parallel. Below you will see how to configure the build to support parallel builds on feature branches. Another very important configuration is the build number, which provides the version for our build. In the build script shown at the end of the chapter, we have some PowerShell variables that pull in variables from the CI build configuration. The build configuration and version are captured here.

$projectConfig = $env:BuildConfiguration

$version = $env:Version

In this way, you can call dotnet.exe so that every DLL is labeled properly. See the command line arguments used as you compile the solution.

Function Compile{

exec {

& dotnet build $syource\_dir\$projectName.sln

-nologo --no-restore -v $verbosity

-maxcpucount --configuration $projectConfig

--no-incremental /p:Version=$version

/p:Authors="Clear Measure"

/p:Product="Onion DevOps Architecture"

}

}

The build script also runs tests that output \*.trx files so that Azure Pipelines can show and track the results of tests as they repeatedly run over time.

Function UnitTests{

Push-Location -Path $unitTestProjectPath

try {

exec {

& dotnet test -nologo -v $verbosity --logger:trx

--results-directory $test\_dir --no-build

--no-restore --configuration $projectConfig

}

}

finally {

Pop-Location

}

}

We are using NUnit as our automated testing framework for this application. Notice that we hard code very little in formulating our commands. This is to make our build script more maintainable. It can also be standardized someone across our teams and other applications given that the variances occur in the properties at the top of the file. Pay special attention to the arguments –no-restore and –no-build. By default, any call to dotnet.exe will recompile your code and perform a Nuget Restore. You do not want to do this, as it is precious time wasted and creates new assemblies just before they are tested.

After the build script finishes, we can run our static analysis tools and then push the application with its various components to Azure Artifacts as \*.nupkg files, which are essentially \*.zip files with some specific differences.

Besides the steps of the build configuration, there are a few other options that should be changed from their defaults. The first is the build number. By default, you have the date embedded as the version number. This can certainly be the default, but to use the SemVer[[6]](#footnote-6), or Semantic Versioning, pattern (<https://semver.org/>), you must change the “Build number format” to the following.

1.0.$(Rev:r).0

Additionally, as you enable continuous integration, you’re asked what branches should be watched. The default is the master branch, but you’ll want to change that to any branch. As you create a branch to develop a backlog item or user story, you’ll want commits on that branch to initiate the pipeline as well. A successful build, deployment, and the full battery of automated tests will give you the confidence that it’s time to put in your pull request. This setting is tricky and not obvious. As you click in the “Branch specification”, you’ll type an asterisk (\*) and hit the Enter key. The figure below shows what you should see.

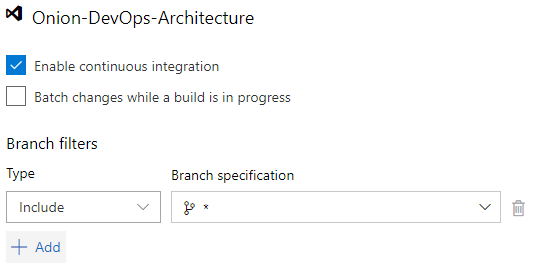


Figure: Configure the continuous integration build to trigger on commits to every branch.

Once your CI build is up and running, add the Build History widget shown below to your project dashboard.

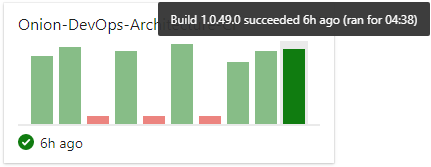


Figure: Seeing the builds on the dashboard can alert you to increasing build times.

Notice that the build time is over four minutes. This is a simple application, but your build time is already up to four minutes and 38 seconds. Yet, your private build runs in about one minute locally. This is because of the hosted build agent architecture. As soon as you have your build stable, you’ll want to start tuning it. One of the first performance optimizations you can make is to attach your own build agent so that you can control the processing power as well as the levels of caching you’d like your build environment to use. Although hosted build agents will certainly improve over time, you must use private build agents in order to achieve the short cycle time necessary to move quickly. And the three minutes overhead you incur at the time of this writing for hosted agents is not what you want for short cycle times across your team.

At the time of this writing, internal Microsoft teams use private build agents in order to achieve the performance and control necessary for complex projects. Use the hosted agents to stabilize new build configurations. Then measure and tune them to decide if you need to provision your own private agents.

Listing: ./build.ps1

. .\BuildFunctions.ps1

$startTime =

$projectName = "OnionDevOpsArchitecture"

$base\_dir = resolve-path .\

$source\_dir = "$base\_dir\src"

$unitTestProjectPath = "$source\_dir\UnitTests"

$integrationTestProjectPath = "$source\_dir\IntegrationTests"

$uiProjectPath = "$source\_dir\UI"

$databaseProjectPath = "$source\_dir\Database"

$projectConfig = $env:BuildConfiguration

$version = $env:Version

$verbosity = "m"

$build\_dir = "$base\_dir\build"

$test\_dir = "$build\_dir\test"

$aliaSql = "$source\_dir\Database\scripts\AliaSql.exe"

$databaseAction = $env:DatabaseAction

if ([string]::IsNullOrEmpty($databaseAction)) { $databaseAction = "Rebuild"}

$databaseName = $env:DatabaseName

if ([string]::IsNullOrEmpty($databaseName)) { $databaseName = $projectName}

$databaseServer = $env:DatabaseServer

if ([string]::IsNullOrEmpty($databaseServer)) { $databaseServer = "localhost\SQL2017"}

$databaseScripts = "$source\_dir\Database\scripts"

if ([string]::IsNullOrEmpty($version)) { $version = "9.9.9"}

if ([string]::IsNullOrEmpty($projectConfig)) {$projectConfig = "Release"}

Function Init {

rd $build\_dir -recurse -force -ErrorAction Ignore

md $build\_dir > $null

exec {

& dotnet clean $source\_dir\$projectName.sln -nologo -v $verbosity

}

exec {

& dotnet restore $source\_dir\$projectName.sln -nologo --interactive

-v $verbosity

}

Write-Host $projectConfig

Write-Host $version

}

Function Compile{

exec {

& dotnet build $source\_dir\$projectName.sln -nologo --no-restore

-v $verbosity

-maxcpucount --configuration $projectConfig --no-incremental

/p:Version=$version /p:Authors="Clear Measure"

/p:Product="Onion DevOps Architecture"

}

}

Function UnitTests{

Push-Location -Path $unitTestProjectPath

try {

exec {

& dotnet test -nologo -v $verbosity --logger:trx

--results-directory $test\_dir

--no-build --no-restore --configuration $projectConfig

}

}

finally {

Pop-Location

}

}

Function IntegrationTest{

Push-Location -Path $integrationTestProjectPath

try {

exec {

& dotnet test -nologo -v $verbosity --logger:trx

--results-directory $test\_dir

--no-build --no-restore --configuration $projectConfig

}

}

finally {

Pop-Location

}

}

Function MigrateDatabaseLocal {

exec{

& $aliaSql $databaseAction $databaseServer $databaseName $databaseScripts

}

}

Function MigrateDatabaseRemote{

$appConfig = "$integrationTestProjectPath\app.config"

$injectedConnectionString = "Server=tcp:$databaseServer,1433

;Initial Catalog=$databaseName

;Persist Security Info=False;User ID=$env:DatabaseUser

;Password=$env:DatabasePassword

;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False

;Connection Timeout=30;"

write-host "Using connection string: $injectedConnectionString"

if ( Test-Path "$appConfig" ) {

poke-xml $appConfig "//add[@key='ConnectionString']/@value"

$injectedConnectionString

}

exec {

& $aliaSql $databaseAction $databaseServer $databaseName $databaseScripts

$env:DatabaseUser

$env:DatabasePassword

}

}

Function Pack{

Write-Output "Packaging nuget packages"

exec{

& .\tools\octopack\Octo.exe pack --id "$projectName.UI"

--version $version

--basePath $uiProjectPath --outFolder $build\_dir

}

exec{

& .\tools\octopack\Octo.exe pack --id "$projectName.Database"

--version $version

--basePath $databaseProjectPath --outFolder $build\_dir

}

exec{

& .\tools\octopack\Octo.exe pack --id "$projectName.IntegrationTests"

--version $version

--basePath $integrationTestProjectPath --outFolder $build\_dir

}

}

Function PrivateBuild{

$sw = [Diagnostics.Stopwatch]::StartNew()

Init

Compile

UnitTests

MigrateDatabaseLocal

IntegrationTest

$sw.Stop()

write-host "Build time: " $sw.Elapsed.ToString()

}

Function CIBuild{

Init

MigrateDatabaseRemote

Compile

UnitTests

IntegrationTest

Pack

}

1. (Duvall, 2007) [↑](#footnote-ref-1)
2. (Beck, Test Driven Development: By Example, 2002) [↑](#footnote-ref-2)
3. (TypeScript in Visual Studio Code, n.d.) [↑](#footnote-ref-3)
4. (Jones, 2012) [↑](#footnote-ref-4)
5. (Humble, 2010) [↑](#footnote-ref-5)
6. (Preston-Werner, n.d.) [↑](#footnote-ref-6)