# Chapter 8: Release Candidate Creation

You have previously learned how to code, build, and validate a build of software. In order for that build to become a release candidate, you must package the build in a format that is suitable for a production release. In addition, in order to make our continuous integration “Commit Stage”[[1]](#footnote-1) more robust, you will have to package the build for release to a TDD (test-driven development) environment. This environment was covered in previous chapters as a deployed environment dedicated solely to the execution of automated tests. The type of tests executed in this environment are Acceptance Tests[[2]](#footnote-2). For a web application, you would commonly use Selenium[[3]](#footnote-3) for tests through the user interface. You can also create other test suites that would require a fully deployed instance of the software system to be deployed. This chapter will focus on the essential elements you will design and configure in order to convert a build to a versioned release candidate suitable for deployment to downstream environments. The chapter will cover the principles involved, the model and relationships of packages to the software architecture, and the process for storing the packages and using them.

## Designing your release candidate architecture

In order to specify the architecture for your release candidate packages, you will need to analyze the logical and physical layers of your 4+1 architecture[[4]](#footnote-4) and determine the unit of deployment. Keep in mind these rules of thumb for release candidate packages:

* **Build/package once, deploy many**  
  A continuous integration build that succeeds will package a set of release candidate packages. These packages should be suitable for deployment to any environment, including production. Do not configure branches or builds so that each environment is built separately. We build and package once so that every subsequent activity performed in our DevOps pipeline further proves the suitability of the release candidate for release to our users of the production environment. When a problem is found, the release candidate is no longer qualified for release. When a release candidate makes its way through our entire pipeline, we can be confident that it is ready for our users.
* **One package per runtime component**Generally, a release candidate is made up of a set of packages (Nuget packages for our .Net applications). Each runtime component in our physical architecture layer makes up a package. That is, our website is packaged entirely in a Nuget package. Our SQL Server database schema and migration scripts are packaged in a separate Nuget package, and so on.
* **Use the Nuget package format**While anyone can invent a new package format, it’s best to use an industry standard format that has tooling such as viewers. The .Net package format is a \*.nupkg file (pronounced NUPKEG). While you could technically put your files in a \*.zip file, you are better served using a more specific format for your platform. Likewise, if you were developing a NodeJS application, you would use the \*.npm package format.
* **Embed the build number as the release candidate version**Build numbers are automatically assigned according to the format you specify. Flow the build number all the way through to each package of your release candidate. If you have build 3.4.352, then your Nuget packages for the release candidate should all be marked 3.4.352. This ensures that regardless of environment, you always know the release candidate being tested and the build that created the release candidate.
* **Package only application artifacts**Take care that you do not package up any environment-specific files or configuration. What is put into each application component package should be suitable for deployment to any environment. Global configuration is appropriate, but any configuration that changes from environment to environment should be pushed in at deploy time, not package time. Be wary of any contents of packages that include reference to an environment name.

### Creating and using release candidate packages

Let’s first look at the part of the process where release candidates fit. The following figure illustrates the sequence of events in our DevOps pipeline.



Figure: Release candidate packages are the bridge between a build and deployment.

The release candidate packages are created by the CI build and are used by the deployment configuration. In Azure Pipelines, you will have a single continuous integration build for your application, regardless of the number of runtime components. For a single Git repository containing a Visual Studio solution, you will have one build. That build, upon success of all the its steps, should package your application for deployment into a set of Nuget packages. We will cover how to determine the number of packages shortly.

Once the release candidate (the results of the build that is about to succeed) is packaged, the build configuration should push them into Azure Artifacts. Azure Artifacts provides a built-in Nuget feed for your team. Unless you have a specific reason to use something else, use this. Once the release candidate packages have been pushed, the CI build finishes and reports success. If the release candidate cannot be packaged or stored, then the build should fail.

While we have not yet covered release configurations or deployments (that will be covered in a later chapter), let’s cover the process that packages are used for. Each of the downstream environments (TDD, UAT, and Prod) will have a deployment process starting with the retrieval of the Nuget packages for the release candidate. This process will access the Nuget feed hosted in Azure Artifacts and retrieve the packages, extract them, and use the contents to deploy your application. The same packages making up a single release candidate should be used for deployment to every environment.

### Defining the bounds of a package

Most applications have more than one runtime component. Therefore, there should be more than one Nuget package in the set of packages that make up a release candidate. Each part of your application that is deployed a specific way should be packaged separately. Let’s dig into how to determine the bounds of each package.



Figure: Each runtime component of the application should have it’s own package.

Let’s consider an application with three deployable components. The first part is a web application using ASP.NET. It is deployed to a web server or to Azure AppService. The second part is an off-line job. Perhaps a batch job or a handler service listening to a queue. These are very common, and they are deployed as Windows Services or Azure Functions or Webjobs. The third is a SQL Server database. Whether on-premise or in Azure SQL, we must deploy schema and global data changes.

Each of these three components of the application run in their own memory space, in their own process, so they have differing deployment characteristics. The deployment destination is different. Because of these, each should be packaged in its own Nuget package. This allows each to be deployed to the appropriate destination while preserving flexibility. While you potentially could deploy all three components to a single server, you could also deploy each to completely different servers. In fact, you could deploy the web application ASP.NET UI to many servers in a web farm or to multiple Azure regions. The structure of your release candidate packages should map to the physical architecture of the running components of the application with a disregard to the topology of the server environment.

In addition to each application component, you will always have some additional assets that go along with the release candidate. These assets are not production code and only exist for validate of the release candidate. In the above figure, you will see “Acceptance Tests”. In Visual Studio, this is manifested as a project, probably NUnit or another testing framework. If the application is a web application, the tests contained in the Visual Studio project likely use Selenium in order to test the fully deployed application. Because of the need for a fully deployed application, these tests are tied to the version of the application; therefore, they belong to a certain release candidate and need to be packaged and deployed along with the other components. In our process diagram earlier in the chapter, you saw that the TDD environment is the first environment for deployments. After the application is fully deployed, the acceptance test package will be retrieved and installed on a deployment server. Then, the tests are run against the deployed release candidate running from the TDD environment. In this way, we can package additional assets for the purpose of making our DevOps pipeline more robust so that it can detect a higher percentage of defects before promoting the release candidate to the next downstream environment.

## Azure Artifacts workflow for release candidates

Now that you have seen how to determine the architecture for your release candidate packaging, let’s see it in action. We will use this configuration for our sample application in Azure DevOps Service. In your project configuration, you will want to make sure you have Azure Artifacts as an enabled service for your pipeline[[5]](#footnote-5). Azure Artifacts is an independent product, but it’s used in conjunction with Azure Pipelines. It’s the storage service for the release candidate components produced by the continuous integration build.

The application we will be packaging has three deployable components that are built and versioned together.

* Website user interface (UI)
* Off-line job
* Database

In addition to these application components, this application also has acceptance tests that will be packaged and deployed. Refer back to chapter 7 to see how a full system acceptance test can be structured when validating a web application. In order to run these acceptance tests against our application in the TDD environment, we must package and store the version of the acceptance tests that belong to this version of the application. The version numbers must match.

Earlier, I stressed how important versioning is in a DevOps pipeline. In the figure below, inspect the release candidate packages in Azure Artifacts.

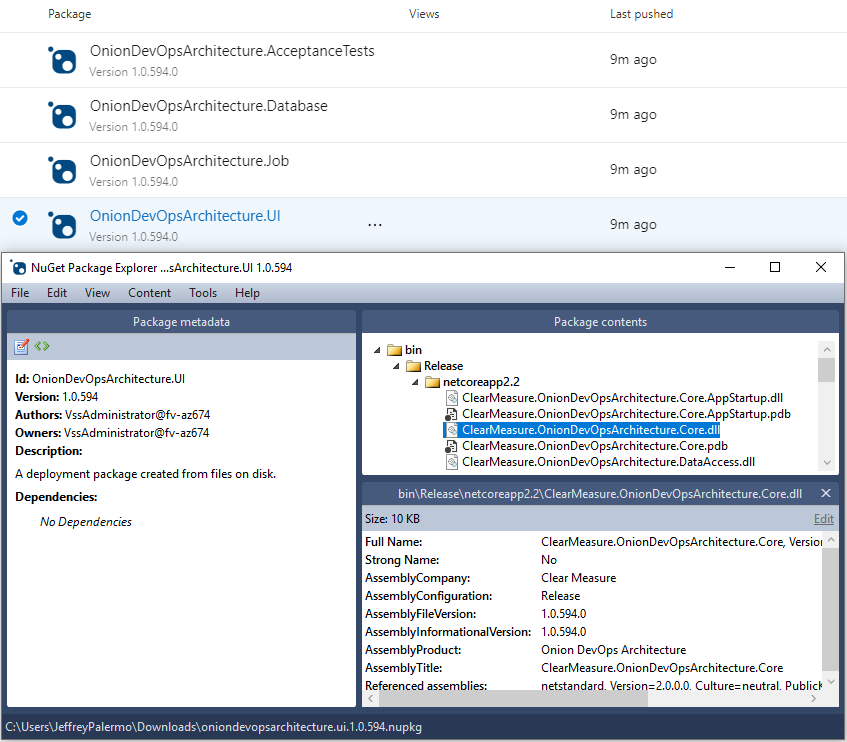


Figure: The version of the release candidate is stamped on the NuGet packages as well as every assembly inside.

Because the proper version number is now embedded into every assembly, your code has access to it. Whether you display it at the bottom of the screen or include it with diagnostics telemetry or logs, you’ll use the version number to know whether a problem or bug was on an old version or the current one. Without the version number, you fly blind. Do not try to use date and time stamps to decipher what build you’re working with. Explicitly push the version number into every asset.

Don’t try to use date and time stamps to decipher what build you’re working with. Explicitly push the version number into every assembly in every release candidate Nuget package.

### Specifying how packages are created

To close the loop on how packages are made, refer back to our build script, which you will want to keep at the top level of your Git repository.

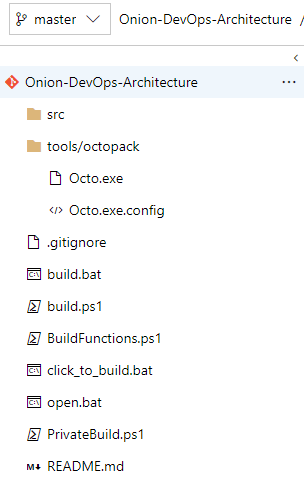


Figure: Our PowerShell build script is stored at the top of the Git repository and is named build.ps1.

When running our private build on the local workstation, packaging is not necessary, but when this script runs as part of the CI build, packaging is the last step before the build should return success. This PowerShell function is responsible for packaging the projects as Nuget packages.

Function Pack{

Write-Output "Packaging nuget packages"

exec{

& dotnet publish $uiProjectPath -nologo --no-restore --no-build

-v $verbosity --configuration $projectConfig

}

exec{

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

--basePath $uiProjectPath\bin\$projectConfig\$framework\publish

--outFolder $build\_dir --overwrite

}

exec{

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

--version $version --basePath $databaseProjectPath

--outFolder $build\_dir --overwrite

}

exec{

& dotnet publish $jobProjectPath -nologo --no-restore --no-build

-v $verbosity --configuration $projectConfig

}

exec{

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

--basePath $jobProjectPath\bin\$projectConfig\$framework\publish

--outFolder $build\_dir --overwrite

}

exec{

& dotnet publish $acceptanceTestProjectPath -nologo --no-restore --no-build

-v $verbosity --configuration $projectConfig

}

exec{

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

--version $version --basePath

$acceptanceTestProjectPath\bin\$projectConfig\$framework\publish

--outFolder $build\_dir --overwrite

}

}

Since we have four components that must be deployed, we have four Nuget packages. Notice that I am using the Octo.exe tool[[6]](#footnote-6). OctoPack is the full name and is an open source wrapper for Nuget, and you can find the source on GitHub. Nuget was originally designed as a package format for library dependencies before it was adapted for application packaging. You can certainly use NuPack directly, but OctoPack wraps NuPack and overcomes some of the default conventions and assumptions that don’t quite fit naturally when packaging an application component for deployment.

With the above PowerShell in our build script, we can configure Azure Pipelines as follows.

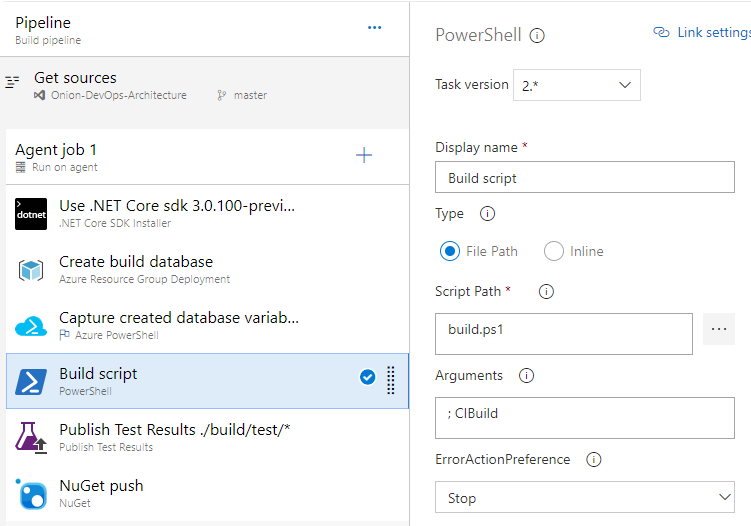


Figure: Azure Pipelines calls the build script stored in Git in order to minimize the global step configuration.

Notice that the “CIBuild” function is being called in the build.ps1 file. This function is as follows.

Function CIBuild{

Init

MigrateDatabaseRemote

Compile

UnitTests

IntegrationTest

Pack

}

The Pack function is the last to be called. Then, with the Nuget packages properly created, our “NuGet push” build step places them in the Azure Artifacts service, which makes them available to any other process that accesses the Nuget feed.

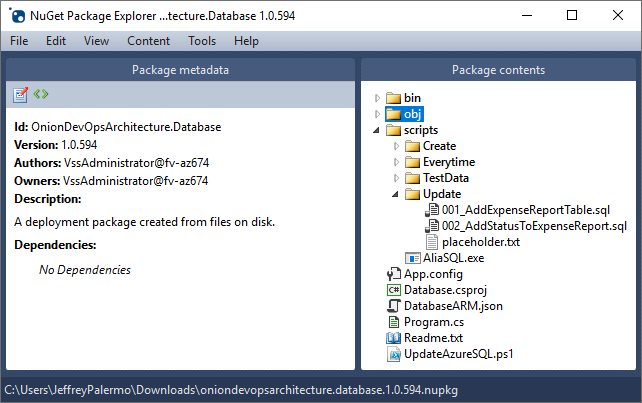


Figure: Our Database component for release candidate 1.0.594 contains the database migration scripts necessary to upgrade the schema.

Because we have the database migration tool and the full body of scripts, we are able to run the database migration process from any server we choose. And including helper \*.ps1 files in our packages will limit the steps that must be explicitly configured in Azure Pipelines.

While YAML will become the format for all builds and releases, the tooling for that capability is not ready yet. Expect to move to YAML-based configurations at the right time. YAML will enable you to store your entire pipeline configuration in your Git repository.

### Use release candidate packages in deployment configurations

While we will go in depth on deploying in the next chapter, it is useful to know how a build or release configuration can retrieve a Nuget package from Azure Artifacts on demand.



Figure: Review of how environment deployments call out to Azure Artifacts to obtain packages.

Remember the order of our environments. In the case of the TDD environment, you will need to deploy the full application and then pull and install the AcceptanceTests package so that those tests can be run. The AcceptanceTests package lives no further than the TDD environment. If you create other types of test suites that must be run on a fully deployed environment, you would use the same pattern of packaging them as a package, storing them in Azure Artifacts, and retrieving them during deploy.

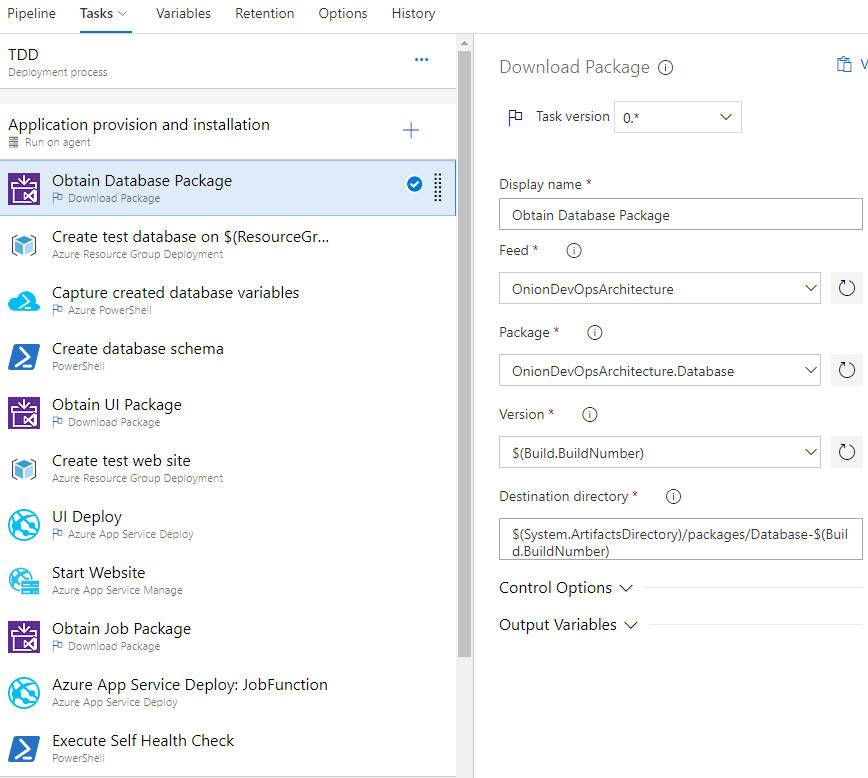


Figure: A portion of the TDD environment release/deploy configuration.

The “Download Package” step allows for easy retrieval of our set of Nuget packages for any release candidate. Because a Nuget package is essentially a \*.zip file with a manifest, this step retrieves and extracts the Nuget package on the destination directory of the server that is used as the Agent, whether it is a Hosted Agent or your own Private Agent. You can see how important the build number is to the pipeline. Everything ties back to the build number.

What about sharing libraries?

If your Visual Studio solution contains a class library that you’d like to share with other teams or other applications, treat it the same as an application component. Package it as part of the CI build. Push it up to Azure Artifacts. Other developers with access that that Nuget feed will be able to pull the library in just like they do with public Nuget packages. And when you have a new build of the library, the feed will automatically receive the library as a result of the new build.

## Wrap up

In this chapter you learned how to package your application as a release candidate, consisting of a set of Nuget packages. Our rules of thumb are:

* Build once, deploy many
* Flow the build number through everything as the official version number
* Use Nuget for your package format
* Archive your release candidates in Azure Artifacts
* Package test suites that must execute in a deployed environment
* Package and publish shared libraries through Azure Artifacts

# Bibliography

(n.d.). Retrieved from OctoPack GitHub Project: https://github.com/OctopusDeploy/OctoPack

(n.d.). Retrieved from Microsoft Docs: Testing with Selenium: https://docs.microsoft.com/en-us/azure/devops/pipelines/test/continuous-test-selenium?view=azure-devops

Dixon, H. (n.d.). *Deep dive into Azure Artifacts*. Retrieved from https://azure.microsoft.com/en-us/blog/deep-dive-into-azure-artifacts/

Duvall, P. M. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison Wesley.

Kruchten, P. (n.d.). Retrieved from Architectural Blueprints—The “4+1” View Model of Software Architecture: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

Wells, D. (n.d.). *Acceptance Tests*. Retrieved 4 3, 2019, from Extremeprogramming.org: http://www.extremeprogramming.org/rules/functionaltests.html

1. (Duvall, 2007) [↑](#footnote-ref-1)
2. (Wells, n.d.) [↑](#footnote-ref-2)
3. (Microsoft Docs: Testing with Selenium, n.d.) [↑](#footnote-ref-3)
4. (Kruchten) [↑](#footnote-ref-4)
5. (Dixon, n.d.) [↑](#footnote-ref-5)
6. (OctoPack GitHub Project, n.d.) [↑](#footnote-ref-6)