# Introduction to Planning Dependencies in your Pipeline

In this module, you'll create a build pipeline that produces a package that can be used by multiple applications.

It's likely that you've used open-source or other third-party components in your software. Using components that are popular in the community and have already been built and tested is often the fastest way to get things done.

You might also have your own application code that you can move into a library or package so that others can use it. This might be an open source project, or software that only your team can access.

There are many ways to build and host your packages. The right solution depends largely on the kinds of programming languages and frameworks you use. Here, you'll continue your work with the Tailspin web team, by creating a NuGet package for .NET that's hosted on Azure Artifacts.

In this module, you will:

* Learn ways to create and share packages that can be used by multiple applications
* Create a build pipeline for your package and publish your package to Azure Artifacts
* Connect an application to your package and build the application in Azure Pipelines
* Push up a change to your package and update your application to use it

## Prerequisites

The modules in this learning path form a progression.

To follow the progression from the beginning, be sure to first complete the Evolve your DevOps practices  learning path.

We also recommend you start at the beginning of this learning path, Build applications with Azure DevOps .

If you want to go through just this module, you need to set up a development environment on your Windows, macOS, or Linux system. You need:

* An Azure DevOps organization
* A GitHub  account
* Visual Studio Code
* .NET Core 3.1 SDK
* Git

You can get started with Azure DevOps for free.

This environment lets you complete the exercises in this and future modules. You can also use it to apply your new skills to your own projects.

**Note**

Keep in mind that you can use Azure DevOps to build and deploy almost any kind of application written in any language. In this module, you'll be working with a .NET Core application written in C#.

You don't need to be an expert in .NET or C# to complete this module. You can apply the patterns you learn here to your own projects that use your favorite programming languages and frameworks.

## Meet the team

You met the Space Game web team at Tailspin Toys in previous modules. As a refresher, here's who you'll work with in this module.



Andy is the development lead.



Amita is in QA.



Tim is in operations.



Mara just joined as a developer and reports to Andy.

Mara has prior experience with DevOps, and is helping the team adopt a more streamlined process by using Azure DevOps.

# Plan build dependencies for your pipeline

In this part, you'll learn about packaging code to make it easier to share. You'll discover why you should make packages, what kinds of packages you can create, where you can host the packages, and how you access them when they are hosted. You'll also learn about package versioning, as well as Azure Artifacts.

But first, let's check in and see how the team is doing. Andy has called the team together to talk about a potential change to their code that would help out another team.

**Andy:** Hi everyone. I was chatting with the team who's working on the back-end system for Space Game. They could use the models we use for the website in a back-end application they plan to write.

**Amita:** What do you mean by models?

**Andy:** As you know, the Space Game website is an ASP.NET Core application. It uses the Model-View-Controller, or MVC, pattern to separate data from how that data is displayed in the user interface. I was thinking we could create a package that contains our model classes so that any application can use them.

**Amita:** What exactly is the goal?

**Andy:** Both of our teams will share the same database. The game sends the database high scores; we read these scores to display on the leaderboard.

**Amita:** That makes sense. How will we create this package?

**Andy:** That's why I wanted to chat with you. We have a few options, and I'm looking for ideas.

**Tim:** I'd love to help, but first I have some questions. I'm new to this and I want to understand how it all works.

## What is a package?

A package contains reusable code that other developers can use in their own projects, even though they didn't write it.

For compiled languages, a package typically contains the compiled binary code, such as .dll files in .NET, or .class files in Java. For languages that are interpreted instead of compiled, such as JavaScript or Python, a package might include source code.

Either way, packages are typically compressed to ZIP or a similar format. Package systems will often define a unique file extension, such as .nupkg or .jar, to make the package's use clear. Compression can help reduce download time, and also produces a single file to make management simpler.

Packages also often contain one or more files that provide metadata, or information, about the package. This metadata might describe what the package does, specify its license terms, the author's contact information, and the package's version.

## Why should I build a package?

There are advantages to building a package as opposed to duplicating the code.

One reason to create a package instead of duplicating code is to prevent drift. When code is duplicated, each copy can quickly diverge to satisfy the requirements of a particular application. It becomes difficult to migrate changes from one copy to the others. In other words, you lose the ability to improve the code in ways that benefit everyone.

Packages also group related functionality into one reusable component. Depending on the programming language, a package can provide applications with access to certain types and functions, while restricting access to their implementation details.

Another reason to build a package is to provide a consistent way to build and test that package's functionality. When code is duplicated, each application might build and test that code in different ways. One set of tests might include checks that another set could benefit from.

One tradeoff is that with a package, you have another codebase to test and maintain. You must also be careful when adding features. Generally speaking, a package should contain features that benefit many kinds of applications. For example, Json.NET  is an extremely popular NuGet package for .NET that enables you to work with JSON files. Json.NET is open source, so the community can propose improvements and report issues.

When multiple applications can benefit from the same code, the advantages far outweigh the disadvantages. You have just one codebase, just one set of tests, and just one build process to manage.

## What kinds of packages are there?

Each programming language or framework provides its own way to build packages. Popular package systems provide documentation on how the process works.

You might already be familiar with these popular package systems:

* **NuGet**: packages .NET libraries
* **Maven**: packages Java libraries
* **npm**: packages JavaScript libraries
* **Chocolatey**: packages Windows applications
* **RubyGems**: packages Ruby libraries

## Where are packages hosted?

You can host packages on your own network, or you can use a hosting service. A hosting service is often called a package repository or package registry. Many of these services provide free hosting for open source projects.

Here are popular hosting services for the package types we just discussed:

* **NuGet**: nuget.org
* **Maven**: apache.org
* **npm**: npmjs.com
* **Chocolatey**: chocolatey.org
* **RubyGems**: rubygems.org

A package feed refers to your package repository server. This server can be on the internet or behind your firewall, on your network. For example, you can host your own NuGet feeds  by using hosting products such as Azure Artifacts and MyGet. You can also host packages on a file share.

When you host packages behind the firewall, you can include feeds to your own packages. You can also cache packages that you trust on your network when your systems can't connect to the internet.

## Who can access packages?

Many package feeds provide unrestricted access to packages. For example, you can download Json.NET  from nuget.org, without the need to sign in or authenticate.

Other package feeds require authentication. You can authenticate access to feeds in a few ways. For example, some kinds of feeds require a username and password. Other feeds require an access token, which is typically a long series of characters that identify who you are and what resources you have access to. Access tokens can be set to expire after a given time period.

## How are packages versioned?

The versioning scheme depends on the packaging system you use.

For example, NuGet packages use Semantic Versioning .

Semantic Versioning is a popular versioning scheme. Here's the format:

**Major.Minor.Patch[-Suffix]**

Here is what each of these parameters means:

* A new **Major** version introduces breaking changes. Applications typically need to update how they use the package to work with a new major version.
* A new **Minor** version introduces new features, but is backward compatible with earlier versions.
* A new **Patch** introduces backward compatible bug fixes, but not new features.

The **-Suffix** part is optional and identifies the package as a pre-release version. For example, 1.0.0-beta1 might identify the package as the first beta pre-release build for the 1.0.0 release.

When you reference a package, you do so by version number.

Here is an example of installing a package by using PowerShell and a specific version number.

**PowerShell**

Install-Package -Name "Json.NET" -Version 12.0.2

## What happens when the package changes?

When you reference a package from your application, you typically pin, or specify, the version of that package you want to use.

Many frameworks enable you to specify allowable ranges of package versions to install. Some also enable you to specify wildcards, which we call a floating version.

For example, in NuGet, version "1.0" means the first version that's equal to or greater than 1.0. "[1.0]" specifies to install version 1.0 only, and not a newer version.

Here are a few other examples:

| **WHAT HAPPENS WHEN THE PACKAGE CHANGES?** | |
| --- | --- |
| **This notation:** | **Selects:** |
| (1.0,) | The first version that's greater than 1. |
| [1.0,2.0] | The first version that's greater than or equal to 1.0, and less than or equal to 2.0 |
| (1.0,2.0) | The first version that's greater than 1.0 and less than 2.0 |
| [1.0,2.0) | The first version that's greater than or equal to 1.0, and less than 2.0 |

As each maintainer releases a new package version, you can evaluate what's changed and test your application against it. When you're ready, you can update the package's version number in your configuration, and submit the change to your build pipeline.

Here's an example of the package dependencies from our Tailspin-SpaceGame-Web project file. In this project, we depend on packages such as Newtonsoft.Json, and you can see that we specify version 12.0.1 of that package.

XMLCopy

<ItemGroup>

<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />

</ItemGroup>

## What is Microsoft Azure Artifacts?

**Mara:** It seems to me it would make sense for us to host the new Models package in Azure Artifacts. We're all part of the Microsoft Azure DevOps organization already, so authentication would be easier than trying to set it up on a different package manager.

**Andy:** I looked into that before the meeting and it seems straightforward to me. I agree with Mara.

**Amita:** What's Azure Artifacts?

**Andy:** Azure Artifacts is a repository in your Azure DevOps organization where you can manage the dependencies for your codebase. Azure Artifacts can store your artifacts and your binaries. It provides a container, called a feed, for groups of dependencies. Developers who have access to the feed can easily consume or publish packages.

## How do I create a package and use it in the pipeline?

**Tim:** So if I am understanding right, the application code uses packages from NuGet already. We're going to create our own package and host it in Azure Artifacts. Can you draw out the pieces and how they'll work together? I'm having a hard time picturing the whole process.

**Andy:** Sure. Let's go over the process of creating a package and using it in our Azure DevOps pipeline.

Andy moves to the whiteboard.

Whiteboard diagram showing the steps to create and use a package

### Create the package

First, we need to create a project in Azure Artifacts. Callout 1 We can do this from Azure DevOps.

Then, we create a pipeline in Azure Pipelines that connects to the GitHub repo for the package code. Then the pipeline builds the code, packages it, and pushes the package to Azure Artifacts. Callout 2

We need to update the application that consumes this package to point to the Azure Artifacts feed that we created. Callout 3

After that, we update the pipeline that creates our application. The update allows us to use our Azure Artifacts feed to pull the new package dependency, and build as normal. Callout 4

### Update the package

**Tim:** What if someone updates the package?

**Andy:** When you update the package with a new feature or bug fix, and run tests to make sure it works correctly, bump up the version number of the package. Then commit the change. The pipeline for the package sees the commit, and creates a new artifact in Azure Artifacts with the new version number. Don't worry, the old package with the lower version number is still there for applications that depend on that version. For this reason, you don't typically unlist a package.

Our application might want to use this newer version of the package. In that case, we update the application to reference the newer version, and run the tests locally to make sure this new version works with our application. When we're satisfied that everything works, we submit the application change to the pipeline. It builds with the new version of the package dependency.

**Amita:** This sounds like a good plan, and it will help the other team too. And it will keep the code from drifting, as you put it. That will help QA as well.

**Mara:** Great! If it's settled, I'll move the code for the package into its own repository.

**Andy:** I'll get started on setting up the Azure Artifacts feed.