AUTO BUILDER

User Documentation v1.8
1/1/2021

CONTENTS

WELCOME

Thanks for checking out **AutoBuilder**! Although there are several different assets on the Asset Store that have similar or overlapping features, nothing out there is as comprehensive or offers a complete workflow solution for asset bundles as **AutoBuilder**. We're proud to offer this package at such a reasonable price, given the number and nature of problems it solves.

If you like the asset, *please* do take a moment to write a positive review or give it a ★★★★★ rating. Be sure to recommend it to your friends. Building assets for the Unity developer community takes a lot of time and effort, so be sure to support the ones that make your life easier. Otherwise we won't be able to keep doing it for you.

The URL to leave a review is right here:

https://assetstore.unity.com/packages/slug/148660

FIRST STEPS (BORING, BUT RECOMMENDED)

This document is the best place to learn about the features and uses about this asset. Please read it thoroughly, as we want to make sure you get the most out of your investment.

1. Create a ***brand new project***. This may seem unnecessary, and while we go through great lengths to make sure there are no collisions with other assets, it is impossible to provide a guarantee for every combination of settings. Overall, this will give you the best possible experience to first figure out how to use the asset without introducing other variables.

2. Install **AutoBuilder** from the Asset Store.

3. Go to the testScenes folder and open the scene called *ABAutoLoader*

    a. Select *ABLoader* from the hierarchy view
    b. Depending on the version of Unity you are using, cut and paste the correct link exactly as you see it:

    Unity 2017:
    https://cdn.reachablegames.com/autobuilderdata/2017/config_{PLATFORM}.json

    Unity 2018:
    https://cdn.reachablegames.com/autobuilderdata/2018/config_{PLATFORM}.json

    Unity 2019:
    https://cdn.reachablegames.com/autobuilderdata/2019/config_{PLATFORM}.json

    Unity 2020:
    https://cdn.reachablegames.com/autobuilderdata/2020/config_{PLATFORM}.json

    Note: Loading asset bundles created by one version of Unity in an application built by a different version of Unity often does not work. Similarly, the Editor can try to load different platform bundles, but they don't always work (like loading Android bundles on Windows in the Unity Editor). My experience is, you really do want the editor version and platform to match what you're loading.

4. Select **Tools→ReachableGames→AutoBuilder→Reveal Build Configs**. This finds the *ABBuildConfig* asset. Select the *ABEditorConfig* a couple of lines down from it.

    a. Set the Build Version to **0.1.12**
    b. Set the platform name to **win32** or **win64** or **osx** depending on your operating system.

    Node: If you are using a version of Unity I don't host test bundles for, or are using Linux, or anything else goes wrong, just run **Tools→ReachableGames→AutoBuilder→Build [Current] for Editor**, and clear the Bootstrap URL. This lets your editor run locally with a set of bundles you just made and put in your local cache, as if they had been downloaded.

5. Press Play. You should see the loading scene pop up a progress bar and it moves smoothly across the screen with the text status changing to indicate what files are being downloaded. A more detailed list of URLs will be written to the Console window, so you can troubleshoot.

6. Once all the files are downloaded, a UI text string says: "Loading From Bundles Starts Now". Three different prefabs are instantiated from the asset bundles that were just retrieved. When all of the cubes are created, a final UI text string says "Completion Callbacks Done". This happens very quickly, and it may appear simultaneous to you… if something is not working, the above is the order it happens in.

7. **Ta-da!** You just ran a project in the Editor that goes through the full asset bundle retrieval, cache management, and object instantiation process, without needing anything more to start with than the C# script assemblies and the *ABEditorConfig* asset to specify the platform and version to retrieve.

Forewarning: These test asset bundles are hosted on my CDN at my expense. As the asset evolves, the version numbers and test scene may change over time, so be sure to keep **AutoBuilder** updated if you plan to try the test scene. This minimal project also works well as an introduction to how to build and host your own set of these asset bundles, as the source assets are provided. You can simply build them and host on your own website or CDN to verify your functionality. Make sure you bump your version number forward to make certain you're not loading a previously cached version from someplace else, of course.

At this point, take a moment to read through the description of the workflows provided by **AutoBuilder** in more detail. Although it is relatively straightforward, there may be details you miss just winging it. The remaining sections will help you gain a deeper understanding of the asset and give instructions on how to use it effectively.

INITIAL CONFIGURATION

Here are the core steps you need to get asset bundle builds working easily and for multiple platforms. **AutoBuilder** is a version-control-friendly system where the configuration of all builds can be checked in and used by anyone, at any time, to generate the next build. The only file that will change because of a build is the version asset, and only if it's set to automatic increment.

1. Create a root bundles folder and organize subfolders into what will become each asset bundle.

2. Configure the platforms for which you plan to build bundles and players in the *ABBuildConfig* asset, and set the root bundles folder appropriately.

3. Configure the *ABBuildVersion* asset the way you plan to use it.

4. Open the *ABAutoLoader* scene and set the name of the scene you want to load after all the asset bundles are cached and mounted.

See *ABNextScene* for an example of how to instantiate objects directly from asset bundles by name. The *testSpawnSomePrefabs.cs* file has a proper async code sample, but I believe there are many ways to use assets in bundles. The hard part is creating bundles in the first place, which **AutoBuilder** does for you.

USING AUTOBUILDER

There are multiple modes for using AutoBuilder. They are essentially configured the same way, but there are additional steps and considerations depending on which mode you want to use. Understanding the differences between use cases will help avoid frustration when assets don't appear to change when the source assets *definitely have changed*. Asset bundles are basically a caching mechanism, so knowing how the cache works (at a high level) is important to be able to debug the caching process.

The absolutely best feature of AutoBuilder is the ability to work in the Editor with asset bundles enabled. This is great, because it means you get truly asynchronous asset loading performance, the ability to profile and performance tune games without having to build a full client and connect to it, and you get to test the same code paths that real built clients will run, rather than a fallback strategy of synchronous loading that works slowly and differently (as is the case with most asset bundle systems).

Why doesn't everyone do this? Normally it means you *must build asset bundles every time you change an asset that is in a bundle*. **That clearly sucks.** AutoBuilder, on the other hand, has a rapid ***Update and Play*** function that you can trigger with a hotkey. This quickly builds a *single override asset bundle* that contains every new asset that is modified based on the current set of asset bundles in use. This takes far less time than building all the bundles again, and before you know it, you're in the game and testing with the new set of assets at full speed.

What's amazing about the Update and Play feature is you can use built bundles, perhaps from a build server, or pull down the most recently published set from a CDN or copy them from the guy sitting next to you. So long as the version of the manifest in the Editor Bundles folder matches the expected version in *ABEditorConfig*, the Update and Play function will generate the override bundle you need to run with all the appropriate assets.

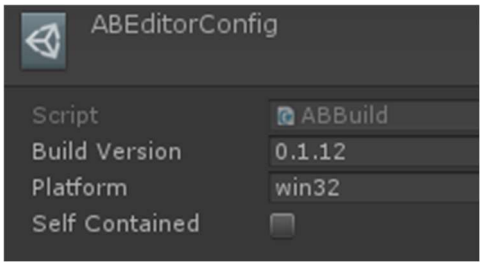
Be aware that if you change a texture that normally lives in an asset bundle and just hit Play (like you are probably used to), **it will not show up** modified. The version in the asset bundle will be loaded instead. It sounds obvious, but when changed assets are "missing" from the game, it's typically because the asset bundles are out of date. Get in the habit of using the shortcut—it'll help tremendously (**Shift-Ctrl-U** on Windows, **Shift-Cmd-U** for Mac). If no bundled assets are changing, just running with Play is perfectly fine, of course.

If many assets have changed since the last full build of asset bundles, it might start taking a while to build the override bundle. Simply rebuild the full asset bundles again and you're back to a rapid iteration time. It's up to you to know the right frequency for this on your project.

EDITOR: LOCAL WORKFLOW

To run completely offline, the process is very simple.

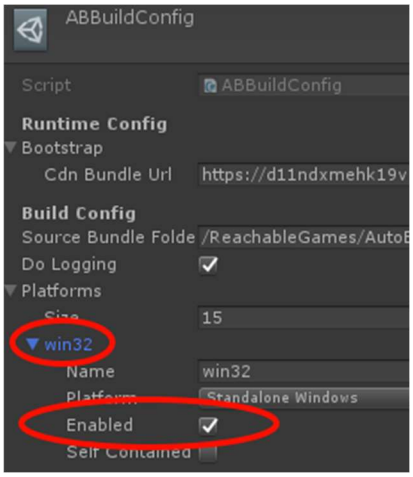
1. Remove any cached data with Tools→ReachableGames→AutoBuilder→**Clear Cached Bundles for Editor**.



2. Go to Tools→ReachableGames→AutoBuilder→**Reveal Build Configs**. Click on ***ABEditorConfig*** in the Project window.

   Set the Platform to whatever yours is (see the Tooltip). The Version string doesn't really matter if you're working locally, but good practice is to set it to the latest version that was published. Leave Self Contained off. There's a different section to describe that below.

3. Edit ***ABBuildConfig***. Make sure your Source Bundle Folder is set properly. Expand the list of *Platforms* and make sure your platform is enabled. It probably is.



4. Build your bundles with Tools→ReachableGames→AutoBuilder→**Build [Current] for Editor**. (These are created directly into your local cache folder, at Tools→ReachableGames→AutoBuilder→**Bundles for Editor**)

5. Open the *ABAutoLoader* scene. Edit the *ABLoader* object in the scene. To work completely offline, clear the Bootstrap URL.

6. Anytime your assets change, launch with Tools→ReachableGames→AutoBuilder→**Update and Play**. You'll need to use this constantly, so there's a keyboard shortcut made for you already.

7. After your assets have changed a lot, the override bundle will start taking longer. Repeat step 4 and it should be fast again. If you're not changing anything that lives in asset bundles, you can just hit Play like normal, of course, but when changes don't seem to apply properly, it's probably because your bundles didn't get rebuilt.

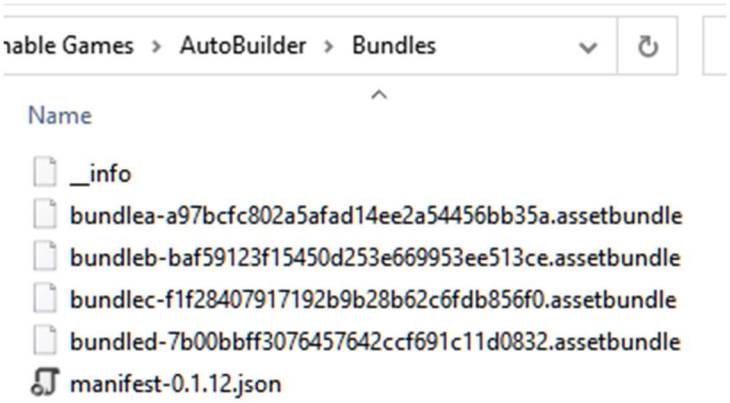
EDITOR: USING HAND-COPIED BUNDLES

To populate your Editor cache with asset bundles that are built by someone else, the process is simple.

1. Run Tools→ReachableGames→AutoBuilder→**Clear Cached Bundles for Editor** on your machine.

2. Open the Tools→ReachableGames→AutoBuilder→**Bundles for Editor** folder on your friend's computer.

3. Copy your friend's bundles into your folder.



4. Edit *ABEditorConfig*
    a. Set Platform to whatever the platform the bundles were built for. See the tooltip for valid choices.
    b. Notice the *manifest-**X.Y.Z**.json* filename. Your **Version** string must be exactly whatever is the X.Y.Z part.

5. Open the *ABAutoLoader* scene. Edit the *ABLoader* object in the scene. To work completely offline, clear the Bootstrap URL.

6. Press *Play*. You're now using your friend's bundles.

7. If you hit *Update and Play*, the editor builds an override bundle with any changes you've got that are different from what is stored in those bundles. Once that is generated, it starts playing and your changes should be visible as well.

EDITOR: USING HOSTED BUNDLES

If you have the option to have a build server generate and automatically publish bundles, this is the easiest option to work with. Of course, publish doesn't necessarily mean public. You should be able to host them internally on a network share drive and use file:// URLs to get at them, or even run a simple web server on your build machine. The rule is, **if you can get to the files in a browser, the game can too**.

1. Edit *ABEditorConfig*
    a. Set **Platform** to whatever the platform the bundles were built for. The string must match a platform string given in the tooltip. Note, the editor can typically load assets from any target (with some limitations), which is convenient for testing across platforms.
    b. Set the **Version** string to the *X.Y.Z* that is in the *manifest-**X.Y.Z**.json* filename.

2. Open the *ABAutoLoader* scene. Edit the *ABLoader* object in the scene.
    a. To work with hosted bundles, enter the URL to where the config files are hosted. As a convenience, *{PLATFORM}* is automatically substituted for the current platform, allowing the same URL configuration to work for all platforms. Example: https://cdn.reachablegames.com/autobuilderdata/config_{PLATFORM}.json

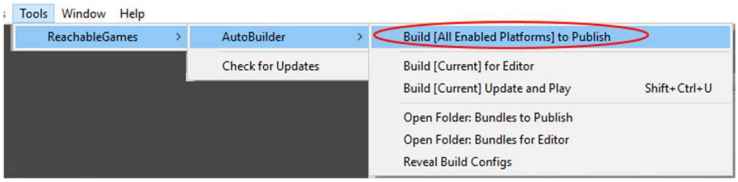
3. Hit Play once to pull down the files locally at least once.

4. Always launch with *Update and Play*. This will create an override bundle with only your changes in it before starting the game.

BUILDING ASSET BUNDLES TO PUBLISH

The build step for making the all the players and asset bundles for publishing builds is very simple.

1. There are two ways to start a multi-platform build.



    a. Simply click on Tools→ReachableGames→AutoBuilder→**Build [All Enabled Platforms] to Publish**
    b. Run builds on the command-line via script with the following command:

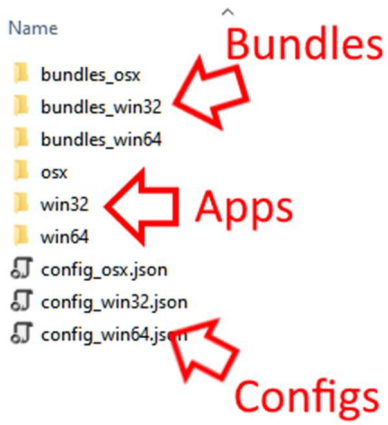
```
You can launch one or multiple builds from the command line like so:
unity.exe -quit -batchmode -projectPath d:\project\path\ -executeMethod
   ReachableGames.AutoBuilder.ABAutoBuilder.BuildFromCommandLine -target=win32 -target=osx

You can launch ALL enabled build platforms like so:
unity.exe -quit -batchmode -projectPath d:\project\path\ -executeMethod
   ReachableGames.AutoBuilder.ABAutoBuilder.BuildFromCommandLine
```

2. If in the Editor, click *Open Folder: Bundles to Publish*. This will take you to the Build folder, which is parallel to your project's Assets folder. These are the files generated by **AutoBuilder** that need to

be put online.

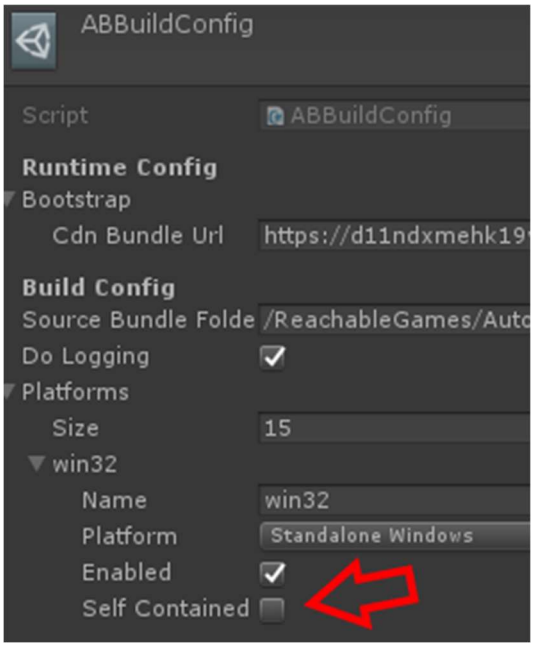
There will be several "Bootstrap" **config_{PLATFORM}.json** files. If you look at their contents, they simply exist to point to where the **bundles_{PLATFORM}** folders can be found. When the game starts up, it only knows how to find the config_win32.json file, for example, then it follows the link stored inside that file to grab the manifest, which lists all the relevant bundles for that version.

Most of the time, you will want to upload both the bundles* and config* files to a web site. It is relatively easy to configure CloudFront as a CDN in front of your web site so you can just keep uploading the files to your web site and they will automatically be propagated to a fast network without any effort. Just make sure your Bootstrap URL in your Unity scene points to the config file on your web site directly, and configure the URL in the ABBuildConfig to point to the CDN.



BUILDING SELF-CONTAINED APPS

A new feature as of v1.6 is Self-Contained builds. Many people have asked for a way to ship the asset bundles along with the game, so that no network access is required. Some platforms might require special permission to access the internet that is off-putting for users. Some developers need to ship a product that doesn't need any kind of network access. Some folks just want to use asset bundles as a way to organize their project without worrying about hosting. Whatever the reason, this feature is for you.

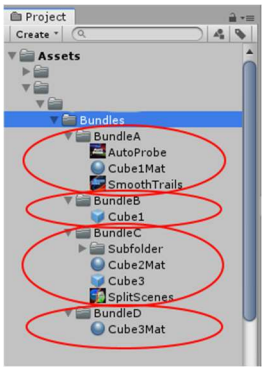


All you have to do is check the **Self Contained** box on the platform you want to build fully self-contained. That's it. When you create a build for that platform, the bundles will built before the executable. They are then moved into the /StreamingAssets/ folder during the executable build process. As soon as the build is complete, they are deleted back out of /StreamingAssets/ so they don't accumulate forever with each new build.

This has been tested successfully on Android, and while I can't guarantee a totally bug-free experience on every platform, Android is by far the hardest to get right. And it works.

On the off-chance you want to debug this specific code path in the Editor, you are welcome to manually copy the config_{PLATFORM}.json file into /StreamingAssets/, and create /StreamingAssets/bundles/ and copy all the contents of your platform bundles in there. Every platform uses the same folder name for simplicity. Then turn on the Self Contained flag in the ABEditorConfig. You lose the ability to *Update and Play*, but to run the debugger on a built set of bundles as closely as possible to the real runtime, this is the way.

ASSET BUNDLE ORGANIZATION



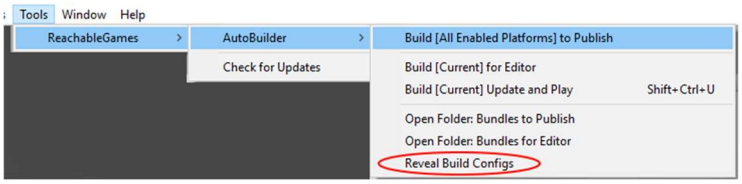
It is worth knowing that bundles can be dependent on assets in other bundles. **AutoBuilder** figures that out automatically for you and prioritizes the load order accordingly. If it can't resolve the ordering, it will error out and tell you the assets that are causing problems. Unity gives many suggestions on ways that asset bundles *might be* used. My experience is that assets with similar modification frequency make good bundle-fellows. It's probably good to have more bundles that are smaller, than to have a few bundles that are larger.

The bundle build process is usually too slow if everything is thrown in a single large bundle, because *anything* that changes requires rebuilding the whole bundle. That requires your users to throw away their massive download and download the new bundle. On the contrary, lots of small bundles that change infrequently is ideal, because most bundles will not need to be rebuilt most of the time. So, make a separate folder for each character or set of characters, for each group of similar props, for each set of related sound effects, etc. The wonderful part about **AutoBuilder** is, you can always just move the files into different folders and the structure of your asset bundles is instantly updated the next time you build without any further effort on your part. You can change your mind anytime.

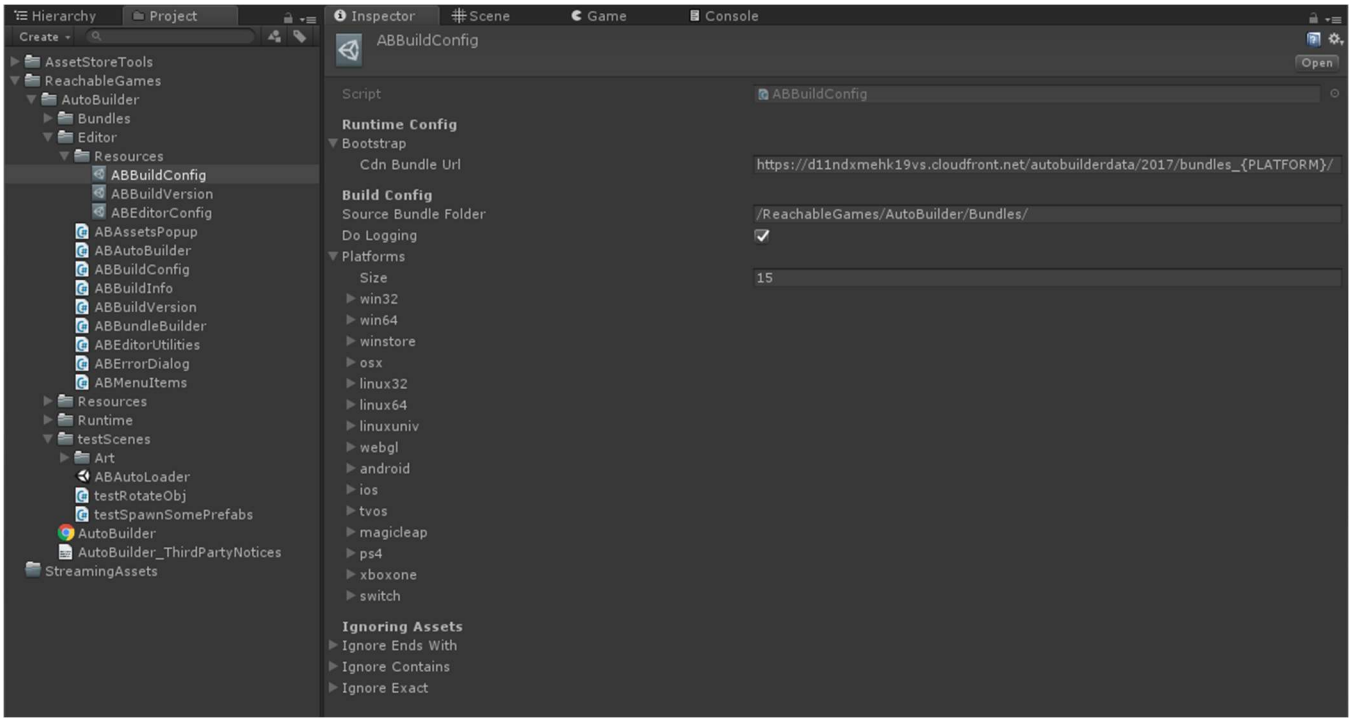
Yes, whole scenes may be made into asset bundles just by moving the scene and its dependencies into a bundle folder. It's possible to load individual assets by name, as is shown in the *testSpawnSomePrefabs.cs* script, but it is not recommended. Unity works best when dependencies are tracked by the system, so rather than loading assets manually by name, make scenes are you normally would and just request those scenes to load as normal. **Scenes will load just fine** once their asset bundles are mounted, all their dependencies included.

A limitation of AssetBundles seems to be Scenes cannot be fully contained within an asset bundle with its dependencies. From what I've observed, a Scene will load fine if all its assets are in another bundle, but if you simply move them all into the same bundle, it fails to load. This is a Unity issue, not an **AutoBuilder** issue.

BUILD CONFIGURATION



Reveal Build Configs will quickly select the *ABBuildConfig* asset for you. This contains the full set of options for each platform that are normally only available in C# for Player and Asset Bundle builds. But with **AutoBuilder**, all you have to do is open the tab for each platform and check some boxes!



- **Source Bundle Folder**
  Use this to specify where the root of your asset bundles folders is. It already assumes /Assets/, so just the rest of the path is necessary.
- **Bootstrap**
  This is a structure that currently only holds a *CDN Bundle URL*, but you could add anything you like, in case there are additional fields that helps your project during startup. **The Bootstrap config is loaded as the very first thing pulled down from the internet on startup**, which is convenient for other data fields, such as URLs for analytics or logging or payments processing, etc. The best reason to put this in a config file is you probably want to have different URLs for things based on whether you're in local development, in a test environment, or a release environment. Baking this into the executable is bad practice, because you would have to re-release a build just to change some web addresses. Instead, move startup configuration details to the config file. Then, when you switch a build from test to release, you just upload a different bootstrap config file.

Upon opening a platform's settings, you are presented with a number of checkboxes that are broken into *Asset Bundles Options* and *Player Build Options* categories.

The ***Enabled*** checkbox controls whether AutoBuilder will try to build the platform. This serves two purposes. One, there is no way to detect a platform is not installed, so you have to remove the platforms that could not build anyway. Two, you might not want to build a platform (yet), but might want to retain the configuration for that platform. Disabling it is a quick and easy way to solve both problems.

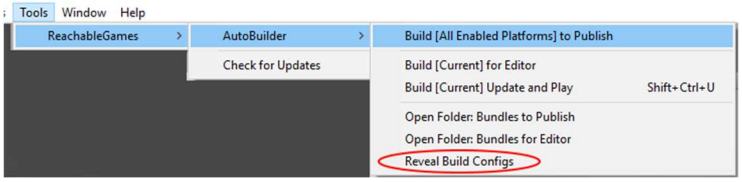
The remaining options are documented on Unity's site. Here is the link to the Asset Bundle Options: https://docs.unity3d.com/ScriptReference/BuildAssetBundleOptions.html

Here is the link to the Player Build Options: https://docs.unity3d.com/ScriptReference/BuildOptions.html

The defaults are generally good to use the way they are set. Different versions of Unity expose different sets of flags, and AutoBuilder exposes them regardless of what version you are using, and just ignores them if they do not apply. Consult the documentation. I do make an effort to ensure that flags that would cause a build to fail (because they are always illegal for that platform, for instance) are automatically disabled, so you don't waste time figuring that stuff out. You're welcome. 😊

N.B. The **Development** flag must be on for any of the other settings in the debugging block to work. It's meaningless to turn on *Connect to Profiler*, for instance, in a non-Development build, so you can simply turn that one off and it turns all the others off as well during the build process.

BUILD VERSION CONFIGURATION



Reveal Build Configs will take you to the folder that contains this asset. The *ABBuildVersion* is a very simple asset that holds Major, Minor, and Build numbers. Major and Minor values are always updated by you, as that should relate to public releases of your product, as is typical with semantic versioning schemes. There are three settings for how you want the Build number to change:

- **Manual**
  When set to Manual, the Build number will only increase if someone changes it explicitly. This is good for script-driven builds or human-driven builds when you aren't producing versions often.
- **AutoIncrement**
  When set to AutoIncrement, the Build number increases every time a Publish build is started, even if it fails to produce an executable. This is because a single version may be used for multiple platform targets, and tracking separate (but functionally equivalent) version numbers per platform is a production nightmare. This setting is easiest to manage for build servers that have the ability to check-in the *ABBuildVersion* asset to revision control after it builds. This is also easy to handle for human-driven builds, and prevents mistakes where accidental modification of existing versions of asset bundles could be catastrophic to your end-user (with a caching CDN, it can create confusion what is being downloaded, so you might not immediately notice).
- **TimeInTicks**
  When set to TimeInTicks, the Build number is always the time in UTC ticks, which almost guarantees unique version numbers for every build, even across a build farm. It also avoids the need to check in the *ABBuildVersion* asset, since the Build value is generated every time anyway.

If you don't like these options, you can always change the *ABBuildVersion* class to do what you want. Its only job is to convert itself to a string, which is then embedded in the *manifest-X.Y.Z.json* filename, and again is embedded in the *ABBuild* asset that gets constructed at build time and embedded in the platform executable, so it knows what version of asset bundles it should load to run correctly. As long as the version is a string of some sort that makes for a valid filename, it should be fine.

Feel free to suggest alternatives, as well. These are just the options we thought would be useful.

FEATURE DETAILS

DESIGN DECISIONS

- **Folders-As-Bundles**
  The whole asset bundle management system (if you can call it that) that Unity has provided is confusing, overly flexible, difficult to visualize, and ultimately hard to manage. Coupled with the fact that it really does take custom coding to make it work well, it's an advanced feature that is out of reach for a lot of smaller teams. By cutting the complexity and offering simple organizational structure and good automated caching, it just works. Each folder becomes an asset bundle, and bundles can depend on other bundles for loading shared assets, giving you the flexibility of just dragging files around to change the organization.
- **Handling Multiple Versions of Same Platform Simultaneously**
  Yes, AutoBuilder is designed to allow multiple versions of each platform to be hosted in the same folder at once. This is extremely helpful if you don't want to or can't invalidate older builds due to certain platform limitations. It is also useful for giving early birds a chance to see the newest build without impacting the rest of the users. Of course, this also means you can publish every version and test with the same infrastructure that you use for releases, and just skip versions that are buggy and release only the ones that pass QA.
- **Build Once, Multiple Environments**
  It is described more thoroughly in the *Bootstrap* section above, but it was a concerted effort to allow for a build process that can promote a single build from an internal development environment all the way to public release, just by changing a web hosted config file. Check out the *Bootstrap* config for details, as it is the first piece of data the *ABAutoLoader* scene fetches, never caches, and is a great place to hook environment settings that differ based on the release status of the project.
- **Filtering Assets by Filename Pattern**
  Some tools generate additional data files that are only needed during the editing process. Unfortunately, not all tools allow us to specify where those files will live, and leave dropping everywhere. Consequently, there are times when filtering out certain files can be critical for reducing asset bundle size. You can set up as many filters are you like, but be aware that they are general filters, not per-folder. This is why exhaustive logging was added, to help determine why files are being filtered out unexpectedly. All slashes are forward, and all paths are lowercase. Some examples:
  - You can use *Ignore Ends With* to strip assets that end with ".png"
  - You can use *Ignore Contains* to strip assets that have "_remove" somewhere in them.
  - You can use *Ignore Exact* to strip an asset like "assets/bundles/folder/filename.png"

FEATURES AUTOBUILDER DOES NOT (AND WILL NOT) HAVE

- **Same-Executable, Updated Asset Bundles**
  This is possible, with a slight change to the code, but not recommended. The way you would do it is set the cache time on the *manifest-X.Y.Z.json* file to be short, like 24 hours, and simply publish over it. The new clients will connect and pull whatever version is hosted and grab the appropriate bundles. However, existing clients cache the manifest, so they will not look at the new hosted file. You would need to change the loader to not cache the manifest. This kind of workflow is poor because there is no easy way to guarantee the state of the executable will be

compatible with data in a set of asset bundles built at a different time. If any scripts change, they are always baked into the executable, but GUIDs which are references to scripts are baked into the assets in bundles. If they don't match, things break. It's safer to publish a new build.

- **Fish**
  Because they are too slimy.

TROUBLESHOOTING

1. Error: Cannot mark assets and scenes in one AssetBundle.

This error pops up whenever Scenes are in the same asset bundles with other kinds of assets. It's not clear why this happens, because it doesn't always pop up, but it's an error coming from within Unity's asset bundle code. If it happens to you, there isn't a lot you can do to fix it except move scene files to a separate bundle.

A good organization that has been suggested is to keep all your scenes in one asset bundle, and keep your assets separate from them. Besides being convenient, the dependencies always work out properly, and it is frequently the case in Unity that scene files change rapidly and assets less rapidly, which makes for better iteration time with *Update and Play*.

2. Materials aren't showing up right, duplicate objects appear tiny in the distance, shaders show up black, etc, **in the editor**.

It seems that Unity's Editor sometimes has a hard time figuring out what shaders to build when using asset bundles, especially when the bundle platform is different from the platform you are using. If you are having problems with it, check that the bundles you are loading were made for your platform, and that they were made with the same version of Unity—although the Editor tries very hard to use all platform data, shaders are not really compatible across all hardware. If all else fails, try bumping your ABBuildVersion and ABEditorConfig to a unique version number and rebuilding the asset bundles locally, to reduce the control some of these variables. **These are visual anomalies in the Editor and are not present on actual built products on real platforms.** Test them on-device before pulling your hair out trying to solve them in the Editor. Rest assured, this is not a bug in AutoBuilder, but problems with the Editor itself.

3. The Cyclical Dependency Error window keeps popping up.

Sorry, you need to move some files around. Either move one or more files into existing bundles where they don't cause cycles in the dependency graph, or create a new bundle and move logically related things into it until there are no more cycles in the dependency graph. The good news is, the text of the warning window tells you exactly which assets are causing problems, and the bundles in which they currently reside.

SUPPORT

Please read all the documentation. We put a lot of effort into it and hope that it exceeds your expectations. In the event you have further questions, please check the following web pages for more details about this asset:

Our Website: https://reachablegames.com/unity-assets/

Unity Forum: https://forum.unity.com/threads/696635/

If you find that none of the above can answer your question, you may contact us at support@reachablegames.com, but know that we always handle support requests first that include:

1. **Invoice Number**
2. **Unity Version**
3. **Asset Version**
4. Links to screenshots or small sample projects on DropBox or similar sharing site describing the problem.
5. Console logs are often helpful.
6. Kindness. If you are mean, rude, harassing, or hateful, do not expect a response.

As a matter of common sense, we do not offer support to free customers except as time permits.

Finally, if you are looking for a feature that is not currently supported, understand that we are a business and get many such requests. If you need a custom feature that is important enough to your product to pay for its development, contact us about it.

ABOUT US

Reachable Games is located in the beautiful hill country of Austin, Texas USA. It was founded by Jason Hughes, who has been a professional game developer since 1995, at Origin Systems. He has worked on many AAA games and with many recognizable companies. As a generalist, Hughes has worked on nearly every kind of game platform in every capacity—from graphics tools to AI to UI/UX to game design to shader writing to database management to networking and server development. It turns out this is the right skillset to help improve the Unity development experience for other developers.

We currently have several other assets on the Asset Store. For what it's worth, they are all built because we are working on projects of our own and both need and use them on a daily basis. If you think this one is great, chances are the others will help you out too.

CHANGE HISTORY

V1.8 – JANUARY 1, 2021

- Revised the error handling for UnityWebRequest now that the interface changed in 2020.2.

V1.7 – OCTOBER 1, 2020

- Minor bug fix in code to avoid C# errors in Unity 2020.x.

V1.6 – SEPTEMBER 30, 2020

- Major new feature: Fully Self-Contained builds is a new checkbox that embeds bundles in the build
- Overhaul of the console logging to help people diagnose where URLs are going wrong.
- Slight refactor of default build folders, test scenes, and hosted bundles. It's a little clearer this way, I think.
- Verified operation in Unity 2020.1.6 and started hosting test bundles for this version.
- Refactored the display in the inspector for the ABBuildConfig.
- Added support for intelligently correcting flags per-platform where they would cause problems or Unity would trigger build failures if set incorrectly.
- Removed the WebHook hackery in favor of popping up a browser tab when updates are released.

V1.5 – JUNE 2, 2020

- Added a product namespace around all the files in AutoBuilder.
- Converted all Assert statements to if/Debug.LogError, since the user experience was poor.
- Fixed an issue with Unity 2019.3 that was causing exceptions deep in C++ that crashed the Editor. Unity's bug, not mine.
- Added feature that allows you to relocate AutoBuilder into any folder without needing to change the code. It just finds itself properly now.
- Created separate links to different test bundles on my CDN. This solves an annoying issue for the test scene and users who have different Unity versions all trying to load bundles from 2017.4LTS.

V1.4 – APRIL 4, 2020

- Fixed error that pops up when an invalid asset (missing script, etc) is added to a bundle, causing builds to fail. Now it adds, but cannot be instantiated. An error will be logged in this case.
- Fixed problem where an edit to an asset would not be picked up by the Override Bundle unless you did Save Project. It works now, by calling Save Project for you.
- Added filename filtering functionality.
- Improved Inspector appearance, added tooltips, etc.
- Added new build options for 2018 and 2019 at the appropriate versions. All options are visible, but have no effect in versions of the editor where it cannot apply them. Consult Unity documentation for details when options are considered valid.

V1.3 – NOV 6, 2019

- Added a cyclical dependency detection system and warning popup dialog, which tells you exactly what assets are connected to other assets in bundles. There's no way we can fix this for you, but with the proper information at hand, the solution is quick—just move one or more files to another bundle, or create a new bundle to break the cycle.
- Added some detail to the documentation, improved screenshots.
- Tracked down some of the shader weirdness in-Editor as being a Unity shader handling bug.
- Pushed a new version of the asset bundles with the Scene in it, to demonstrate that flow.

V1.2 – AUG 29, 2019

- Updated the code to work in Unity 2017.4 and newer, which required a few changes.
- Improved legibility of the colored error messages.

V1.1 – JULY 8, 2019

- Fixed a bug with UnityEditor.SceneAsset not being handled correctly when generating the dynamic type (because it doesn't exist at runtime, truly). This caused some errors while reading the manifest when a scene was the only thing in a bundle.

V1.0 – JULY 4, 2019

- First release.