ue4-docker user manual

Adam Rehn, other contributors

2022-12-13

Table of Contents

Read these first

Introduction to ue4-docker

The ue4-docker Python package contains a set of Dockerfiles and accompanying build infrastructure that allows you to build Docker images for Epic Games' Unreal Engine. The images also incorporate the infrastructure from `ue4cli`, `conan-ue4cli`, and `ue4-ci-helpers` to facilitate a wide variety of use cases.

Key features include:

- Unreal Engine 4.20.0 and newer is supported.
- Both Windows containers and Linux containers are supported.
- Building and packaging Unreal Engine projects is supported.
- Running automation tests is supported.
- Running built Unreal Engine projects with offscreen rendering is supported via the NVIDIA Container Toolkit under Linux.

Important legal notice

Except for the ue4-build-prerequisites image, the Docker images produced by the ue4-docker Python package contain the Unreal Engine Tools in both source code and object code form. As per Section 1A of the Unreal Engine EULA, Engine Licensees are prohibited from public distribution of the Engine Tools unless such distribution takes place via the Unreal Marketplace or a fork of the Epic Games Unreal Engine GitHub repository. Public distribution of the built images via an openly accessible Docker Registry (e.g. Docker Hub) is a direct violation of the license terms. **It is your responsibility to ensure that any private distribution to other Engine Licensees (such as via an organisation's internal Docker Registry) complies with the terms of the Unreal Engine EULA.**

For more details, see the Unreal Engine EULA Restrictions page on the Unreal Containers community hub.

Getting started

Multipurpose Docker images for large, cross-platform projects such as the Unreal Engine involve a great deal more complexity than most typical Docker images. Before you start using ue4-docker it may be helpful to familiarise yourself with some of these complexities:

- If you've never built large (multi-gigabyte) Docker images before, be sure to read the Large container images primer.
- If you've never used Windows containers before, be sure to read the Windows containers primer.
- If you've never used GPU-accelerated Linux containers with the NVIDIA Container Toolkit before, be sure to read the NVIDIA Container Toolkit primer.

Once you're familiar with all the relevant background material, you can dive right in:

1. First up, head to the Configuration section for details on how to install ue4-docker and configure your host system so that it is ready to build and run the Docker images.
2. Next, check out the Use Cases section for details on the various scenarios in which the Docker images can be used. Once you've selected the use case you're interested in, you'll find step-by-step instructions on how to build the necessary container images and start using them.
3. If you run into any issues or want to customise your build with advanced options, the Building Images section provides all the relevant details.
4. For more information, check out the FAQ and the Command Reference section.

Links

- ue4-docker GitHub repository
- ue4-docker package on PyPI
- Related articles on adamrehn.com
- Unreal Containers community hub

Large container images primer

Although large container images are in no way different to smaller container images at a technical level, there are several aspects of the Docker build process that impact large images to a far greater extent than smaller images. This page provides an overview for users who have never built large (multi-gigabyte) container images before and may therefore be unfamiliar with these impacts. This information applies equally to both Linux containers and Windows containers.

Filesystem layer commit performance

The time taken to commit filesystem layers to disk when building smaller Docker images is low enough that many users may not even perceive this process as a distinct aspect of a `RUN` step in a Dockerfile. However, when a step generates a filesystem layer that is multiple gigabytes in size, the time taken to commit this data to disk becomes immediately noticeable. For some larger layers in the container images built by ue4-docker, the filesystem layer commit process can take well over 40 minutes to complete on consumer-grade hardware. (The Installed Build layer in the multi-stage build of the ue4-minimal image is the largest of all the filesystem layers, and has been observed to take well over an hour and a half to commit to disk on some hardware.)

Since Docker does not emit any output during the layer commit process, users may become concerned that the build has hung. After the ue4-docker provided output `Performing filesystem layer commit…`, the only indication that any processing is taking place is the high quantity of CPU usage and disk I/O present during this stage. There is no need for concern, as none of the steps in the ue4-docker Dockerfiles can run indefinitely without failing and emitting an error. When a build step ceases to produce output, it is merely a matter of waiting for the filesystem layer commit to complete.

Disk space consumption during the build process

Due to overheads associated with temporary layers in multi-stage builds and layer difference computation, the Docker build process for an image will consume more disk space than is required to hold the final built image. These overheads are relatively modest when building smaller container images. However, these overheads are exacerbated significantly when building large container images, and it is important to be aware of the quantity of available disk space that is required to build any given image or set of images.

Although none of the container images produced by ue4-docker currently exceed 100GB in size, the build process requires at least 400GB of available disk space under Linux and at least 800GB of available disk space under Windows. Once a build is complete, the ue4-docker clean command can be used to clean up temporary layers leftover from multi-stage builds and reclaim all the disk space not occupied by the final built images. The docker system prune command can also be useful for cleaning up data that is not used by any of the tagged images present on the system.

Windows containers primer

The implementation-agnostic information from this page has migrated to the Unreal Containers community hub. You can find the new version here: Key Concepts: Windows Containers.

Details specific to ue4-docker:

Due to the performance and stability issues currently associated with containers running in Hyper-V isolation mode, it is strongly recommended that process isolation mode be used for building and running Windows containers. This necessitates the use of Windows Server as the host system (or Windows 10 version 1809 or newer for development and testing purposes) and requires that all container images use the same Windows version as the host system. A number of ue4-docker commands provide specific functionality to facilitate this:

- The ue4-docker build command will automatically attempt to build images based on the same kernel version as the host system, and will default to process isolation mode if the operating system version and Docker daemon version allow it. Hyper-V isolation mode will still be used if the user explicitly specifies a different kernel version than that of the host system or explicitly requests Hyper-V isolation mode.
- The ue4-docker setup command automates the configuration of Windows Server hosts, in order to provide a smoother experience for users who migrate their container hosts to the latest versions of Windows Server as they are released.

NVIDIA Container Toolkit primer

This page has migrated to the Unreal Containers community hub. You can find the new version here: Key Concepts: NVIDIA Container Toolkit.

Configuration

Supported host configurations

The table below lists the host operating systems can be used to build and run the container images produced by ue4-docker, as well as which features are supported under each system.

Click on an operating system's name to view the configuration instructions for that platform.

Host OS	Linux containers	Windows containers	NVIDIA Container Toolkit	Optimality
Linux	✔	✖	✔	Optimal for Linux containers
Windows Server	✖	✔	✖	Optimal for Windows containers when using process isolation mode
Windows 10 and 11	Works but not tested or supported	✔	✖	Optimal for Windows containers when using process isolation mode
macOS	✔	✖	✖	Suboptimal for Linux containers

The **Optimality** column indicates whether a given host operating system provides the best experience for running the container types that it supports. The configuration instructions page for each operating system provides further details regarding the factors that make it either optimal or suboptimal.

Configuring Linux

Requirements

- 64-bit version of one of Docker's supported Linux distributions (CentOS 7+, Debian 7.7+, Fedora 26+, Ubuntu 14.04+)
- Minimum 8GB of RAM
- Minimum 400GB available disk space for building container images

Step 1: Install Docker CE

Follow the official installation instructions from the Docker Documentation for your distribution:

- CentOS
- Debian
- Fedora
- Ubuntu

Once Docker is installed, follow the instructions from the Post-installation steps for Linux page of

the Docker Documentation to allow Docker commands to be run by a non-root user. This step is required in order to enable audio support when performing cloud rendering using the NVIDIA Container Toolkit.

Step 2: Install Python 3.6 or newer

Note that older versions of these Linux distributions may not have Python 3.6 available in their system repositories by default. When working with an older distribution it may be necessary to configure community repositories that provide newer versions of Python.

Under CentOS, run:

```
sudo yum install python3 python3-devel python3-pip
```

Under Debian and Ubuntu, run:

```
sudo apt-get install python3 python3-dev python3-pip
```

Under Fedora, run:

```
sudo dnf install python3 python3-devel python3-pip
```

Step 3: Install ue4-docker

Install the ue4-docker Python package by running the following command:

```
sudo pip3 install ue4-docker
```

Step 4: Use ue4-docker to automatically configure the Linux firewall

If the host system is running an active firewall that blocks access to port 9876 (required during the build of the ue4-source image) then it is necessary to create a firewall rule to permit access to this port. The ue4-docker setup command will detect this scenario and perform the appropriate firewall configuration automatically. Simply run:

```
sudo ue4-docker setup
```

Note that the `iptables-persistent` service will need to be installed for the newly-created firewall rule to persist after the host system reboots.

Configuring Windows Server

Windows Server provides an optimal experience when running Windows containers, but **only when process isolation mode is used**. Using Hyper-V isolation mode will result in a suboptimal experience due to several issues that impact performance and stability. Process isolation mode is the default isolation mode under Windows Server.

Requirements

- Windows Server 2016 or newer
- Minimum 8GB of RAM
- Minimum 800GB available disk space for building container images

Step 1: Install Docker EE

As per the instructions provided by the Install Docker Engine - Enterprise on Windows Servers page of the Docker Documentation, run the following commands from an elevated PowerShell prompt:

```
# Add the Docker provider to the PowerShell package manager
Install-Module DockerMsftProvider -Force

# Install Docker EE
Install-Package Docker -ProviderName DockerMsftProvider -Force

# Restart the computer to enable the containers feature
Restart-Computer
```

Step 2: Install Python 3 via Chocolatey

The simplest way to install Python and pip under Windows is to use the Chocolatey package manager. To do so, run the following command from an elevated PowerShell prompt:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager
]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
iex ((New-Object System.Net.WebClient).DownloadString
('https://community.chocolatey.org/install.ps1'))
```

You may need to restart the system for your shell to recognise the updates that the Chocolatey installer makes to the system `PATH` environment variable. Once these changes are recognised, you can install Python by running the following command from either an elevated PowerShell prompt or an elevated Command Prompt:

```
choco install -y python
```

Step 3: Install ue4-docker

Install the ue4-docker Python package by running the following command from an elevated Command Prompt:

```
pip install ue4-docker
```

Step 4: Use ue4-docker to automatically configure Docker and Windows Firewall

To automatically configure the required system settings, run the ue4-docker setup command from an elevated Command Prompt:

```
ue4-docker setup
```

This will configure the Docker daemon to set the maximum image size to 400GB, create a Windows Firewall rule to allow Docker containers to communicate with the host system (which is required during the build of the ue4-source image), and download any required DLL files under Windows Server version 1809 and newer.

Configuring Windows 10 and 11

Warning

Windows 10 and 11 provide an optimal experience when running Windows containers, but **only when process isolation mode is used**. Using Hyper-V isolation mode will result in a suboptimal experience due to several issues that impact performance and stability. The default isolation mode depends on the specific version of Windows being used:

- Under Windows 10, Hyper-V isolation mode is the default isolation mode and process isolation mode must be manually enabled each time a container is built or run. The ue4-docker build command will automatically pass the flag to enable process isolation mode where possible. **This requires Windows 10 version 1809 or newer.**
- Under Windows 11, process isolation mode is the default isolation mode.

Requirements

- 64-bit Windows 10 Pro, Enterprise, or Education (Version 1607 or newer)
- Hardware-accelerated virtualization enabled in the system BIOS/EFI
- Minimum 8GB of RAM
- Minimum 800GB available disk space for building container images

Step 1: Install Docker CE for Windows

Download and install Docker CE for Windows from the Docker Store.

Step 2: Install Python 3 via Chocolatey

The simplest way to install Python and pip under Windows is to use the Chocolatey package manager. To do so, run the following command from an elevated PowerShell prompt:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager
]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
iex ((New-Object System.Net.WebClient).DownloadString
('https://community.chocolatey.org/install.ps1'))
```

You may need to restart your shell for it to recognise the updates that the Chocolatey installer makes to the system `PATH` environment variable.

Once these changes are recognised, you can install Python by running the following command from either an elevated PowerShell prompt or an elevated Command Prompt:

```
choco install -y python
```

Step 3: Install ue4-docker

Install the ue4-docker Python package by running the following command from an elevated Command Prompt:

```
pip install ue4-docker
```

Step 4: Manually configure Docker daemon settings

For building and running Windows containers:

- Configure the Docker daemon to use Windows containers rather than Linux containers.
- Configure the Docker daemon to increase the maximum container disk size from the default 20GB limit by following the instructions provided by Microsoft. The 120GB limit specified in the instructions is not quite enough, so set a 400GB limit instead. **Be sure to restart the Docker daemon after applying the changes to ensure they take effect.**

The ue4-docker maintainers do not provide support for building and running Linux containers under Windows, due to the various technical limitations of the Hyper-V and WSL2 backends for Docker Desktop (see this issue for details of these limitations). This functionality is still present in ue4-docker for those who choose to use it, but users are solely responsible for troubleshooting any issues they encounter when doing so.

For building and running Linux containers:

- Configure the Docker daemon to use Linux containers rather than Windows containers.
- **If you are using the Hyper-V backend** then use the Advanced section under the Resources tab of the Docker Desktop settings pane to set the memory allocation for the Moby VM to 8GB and the maximum VM disk image size to 400GB.
- **If you are using the WSL2 backend** then expand the WSL2 virtual hard disk to at least 400GB.

Configuring macOS

macOS provides a suboptimal experience when running Linux containers, due to the following factors: Linux containers are unable to use GPU acceleration via the NVIDIA Container Toolkit.

Requirements

- 2010 or newer model Mac hardware
- macOS 10.10.3 Yosemite or newer
- Minimum 8GB of RAM
- Minimum 400GB available disk space for building container images

Step 1: Install Docker CE for Mac

Download and install Docker CE for Mac from the Docker Store.

Step 2: Install Python 3 via Homebrew

The simplest way to install Python 3 and pip under macOS is to use the Homebrew package manager. To do so, run the following commands from a Terminal prompt:

```
# Install Homebrew
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Install Python

```
brew install python
```

Step 3: Install ue4-docker

Install the ue4-docker Python package by running the following command from a Terminal prompt:

```
sudo pip3 install ue4-docker
```

Step 4: Manually configure Docker daemon settings

Use the Advanced section under the Resources tab of the Docker Desktop settings pane to set the memory allocation for the Moby VM to 8GB and the maximum VM disk image size to 400GB.

Use cases

Use cases overview

The container images produced by ue4-docker incorporate infrastructure to facilitate a wide variety of use cases. A number of key use cases are listed below. Select a use case to see detailed instructions on how to build and run the appropriate container images.

- Continuous Integration
- Linux installed builds
- Microservices

Continuous Integration (CI)

Use a controlled and reproducible environment to build, test, and package Unreal projects.

This page has migrated to the Unreal Containers community hub. You can find the new version here: Use Cases: Continuous Integration and Deployment (CI/CD).

Microservices

Thanks to the inclusion of conan-ue4cli infrastructure, the ue4-full image makes it easy to build Unreal Engine-powered microservices with Google's popular gRPC framework.

This page has migrated to the Unreal Containers community hub. You can find the new version here: Use Cases: Microservices.

Linux Installed Builds

Under Windows and macOS, Engine licensees can easily download and manage Installed Builds of Unreal Engine through the Epic Games Launcher. Since version 4.21.0 of the Engine, ue4-docker provides an alternative source of Installed Builds under Linux in lieu of a native version of the launcher.

This page has migrated to the Unreal Containers community hub. You can find the new version here: Use Cases: Linux Installed Builds.

Building images

List of available container images

You can build the following images using the ue4-docker build command:

- ue4-build-prerequisites
- ue4-source
- ue4-minimal
- ue4-full

By default, all available images will be built. You can prevent unwanted images from being built by appending the relevant image-specific flag to the build command (see the **"Flag to disable"** entry for each image below.)

ue4-build-prerequisites

Tags:

- **Windows containers**: `adamrehn/ue4-build-prerequisites:BASETAG` where `BASETAG` is the Windows Server Core base image tag
- **Linux containers**: `adamrehn/ue4-build-prerequisites:CONFIGURATION` where `CONFIGURATION` is as follows:
  - `opengl` if CUDA support is not enabled
  - `cudaglVERSION` where `VERSION` is the CUDA version if CUDA support is enabled (e.g. `cudagl9.2`, `cudagl10.0`, etc.)

Dockerfiles: Windows | Linux

Contents: Contains the build prerequisites common to all Engine versions.

Uses:

- Keep this image on disk to speed up subsequent container image builds.

ue4-source

Tags:

- `adamrehn/ue4-source:RELEASE` where `RELEASE` is the Engine release number
- `adamrehn/ue4-source:RELEASE-PREREQS` where `RELEASE` is as above and `PREREQS` is the ue4-build-prerequisites image tag

Dockerfiles: Windows | Linux

Contents: Contains the cloned source code for Unreal Engine, along with its downloaded

dependency data. The ue4-minimal image uses this source code as the starting point for its build.

Uses:

- Only needed during the build process. Afterwards, this image can be removed using `ue4-docker clean --source` to save disk space.

ue4-minimal

Tags:

- `adamrehn/ue4-minimal:RELEASE` where `RELEASE` is the Engine release number
- `adamrehn/ue4-minimal:RELEASE-PREREQS` where `RELEASE` is as above and `PREREQS` is the ue4-build-prerequisites image tag

Dockerfiles: Windows | Linux

Contents: Contains the absolute minimum set of components required for use in a Continuous Integration (CI) pipeline, consisting of only the build prerequisites and an Installed Build of the Engine.

Uses:

- Use this image for CI pipelines that do not require ue4cli, conan-ue4cli, or ue4-ci-helpers.

ue4-full

Tags:

- `adamrehn/ue4-full:RELEASE` where `RELEASE` is the Engine release number
- `adamrehn/ue4-full:RELEASE-PREREQS` where `RELEASE` is as above and `PREREQS` is the ue4-build-prerequisites image tag

Dockerfiles: Windows | Linux

Contents: Contains everything from the `ue4-minimal` image, and adds the following:

- ue4cli
- conan-ue4cli
- ue4-ci-helpers
- PulseAudio support (Linux image only)
- X11 support (Linux image only)

Uses:

- CI pipelines that require ue4cli, conan-ue4cli, or ue4-ci-helpers
- Packaging Unreal Engine-powered microservices

Advanced build options

General options

Specifying Git credentials

The ue4-docker build command supports three methods for specifying the credentials that will be used to clone the Unreal Engine Git repository:

- **Command-line arguments**: the `-username` and `-password` command-line arguments can be used to specify the username and password, respectively.
- **Environment variables**: the `UE4DOCKER_USERNAME` and `UE4DOCKER_PASSWORD` environment variables can be used to specify the username and password, respectively. Note that credentials specified via command-line arguments will take precedence over values defined in environment variables.
- **Standard input**: if either the username or password has not been specified via a command-line argument or environment variable then the build command will prompt the user to enter the credential(s) for which values have not already been specified.

Note that the username and password are handled independently, which means you can use different methods to specify the two credentials (e.g. username specified via command-line argument and password supplied via standard input.)

Users who have enabled Two-Factor Authentication (2FA) for their GitHub account will need to generate a personal access token and use that in place of their password.

Building a custom version of the Unreal Engine

If you would like to build a custom version of Unreal Engine rather than one of the official releases from Epic, you can specify "custom" as the release string and specify the Git repository and branch/tag that should be cloned. When building a custom Engine version, **both the repository URL and branch/tag must be specified**:

If you would like to build a custom version of Unreal Engine rather than one of the official releases from Epic, you can specify "custom" as the release string and specify the Git repository and branch/tag that should be cloned. When building a custom Engine version, **both the repository URL and branch/tag must be specified**:

```
ue4-docker build custom -repo=https://github.com/MyUser/UnrealEngine.git -branch
=MyBranch
```

This will produce images tagged `adamrehn/ue4-source:custom`, `adamrehn/ue4-minimal:custom`, etc.

If you are performing multiple custom builds and wish to differentiate between them, it is recommended to also specify a name for the custom build:

```
ue4-docker build custom:my-custom-build -repo
```

```
=https://github.com/MyUser/UnrealEngine.git -branch=MyBranch
```

This will produce images tagged `adamrehn/ue4-source:my-custom-build`, `adamrehn/ue4-minimal:my-custom-build`, etc.

Excluding Engine components to reduce the final image size

Starting in ue4-docker version 0.0.30, you can use the `--exclude` flag when running the ue4-docker build command to specify that certain Engine components should be excluded from the ue4-minimal and ue4-full images. The following components can be excluded:

- `ddc`: disables building the DDC for the Engine. This significantly speeds up building the Engine itself but results in far longer cook times when subsequently packaging Unreal projects.
- `debug`: removes all debug symbols from the built images. (When building Windows containers the files are actually truncated instead of removed, so they still exist but have a size of zero bytes. This is done for compatibility reasons.)
- `templates`: removes the template projects and samples that ship with the Engine.

You can specify the `--exclude` flag multiple times to exclude as many components as you like. For example:

```
# Excludes both debug symbols and template projects
ue4-docker build 4.21.2 --exclude debug --exclude templates
```

Enabling system resource monitoring during builds

Starting in ue4-docker version 0.0.46, you can use the `--monitor` flag to enable a background thread that will log information about system resource usage (available disk space and memory, CPU usage, etc.) at intervals during the build. You can also use the `-interval` flag to override the default interval of 20 seconds:

```
# Logs system resource levels every 20 seconds
ue4-docker build 4.24.2 --monitor
```


```
# Logs system resource levels every 20 seconds
ue4-docker build 4.24.2 --monitor
```


```
# Logs system resource levels every 5 seconds
ue4-docker build 4.24.2 --monitor -interval=5
```

Exporting generated Dockerfiles

Since ue4-docker version 0.0.78, the ue4-docker build command supports a flag called `-layout` that allows the generated Dockerfiles to be exported to a filesystem directory instead of being built. In

addition, version 0.0.80 of ue4-docker added support for a flag called `--combine` that allows you to combine multiple generated Dockerfiles into a single Dockerfile that performs a multi-stage build. You can use these flags like so:

```
# Exports Dockerfiles for all images to the specified filesystem directory
ue4-docker build 4.25.4 -layout "/path/to/Dockerfiles"
```

```
# Exports Dockerfiles for all images
ue4-docker build 4.25.4 -layout "/path/to/Dockerfiles"
```

```
# Exports Dockerfiles for all images and combines them into a single Dockerfile
ue4-docker build 4.25.4 -layout "/path/to/Dockerfiles" --combine
```

Exporting Dockerfiles is useful for debugging or contributing to the development of ue4-docker itself. You can also use the generated Dockerfiles to build container images independently of ue4-docker, but only under the following circumstances:

- When building Windows container images, you must specify the advanced option `source_mode` and set it to `copy`. This generates Dockerfiles that copy the Unreal Engine source code from the host filesystem rather than cloning it from a git repository, thus eliminating the dependency on ue4-docker's credential endpoint to securely provide git credentials and allowing container images to be built without the need for ue4-docker itself.
- When building Linux container images, you must either set the advanced option `source_mode` to `copy` as detailed above, or else specify the `credential_mode` option and set it to `secrets`. This generates Dockerfiles that use the Linux-only BuildKit build secrets functionality to securely provide git credentials, eliminating the dependency on ue4-docker's credential endpoint whilst still facilitating the use of a git repository to provide the Unreal Engine source code.

Advanced options for Dockerfile generation

Note that option names are all listed with underscores between words below (e.g. `source_mode`), but in some examples you will see dashes used as the delimiter instead (e.g. `source-mode`). **These uses are actually equivalent, since ue4-docker automatically converts any dashes in the option name into underscores.** This is because dashes are more stylistically consistent with command-line flags (and thus preferable in examples), but underscores must be used in the underlying Dockerfile template code since dashes cannot be used in Jinja identifiers.

Since ue4-docker version 0.0.78, the ue4-docker build command supports a flag called `--opt` that allows users to directly set the context values passed to the underlying Jinja templating engine used to generate Dockerfiles. Some of these options (such as `source_mode`) can only be used when exporting generated Dockerfiles, whereas others can be used with the regular ue4-docker build process. **Note that incorrect use of these options can break build behaviour, so only use an option if you have read through both this documentation and the ue4-docker source code itself and understand exactly what that option does.** The following options are supported as of

the latest version of ue4-docker:

- `source_mode`: **(string)** controls how the ue4-source Dockerfile obtains the source code for the Unreal Engine. Valid options are:
- `git`: the default mode, whereby the Unreal Engine source code is cloned from a git repository. This is the only mode that can be used when not exporting generated Dockerfiles.
- `copy`: copies the Unreal Engine source code from the host filesystem. The filesystem path can be specified using the `SOURCE_LOCATION` Docker build argument, and of course must be a child path of the build context.
- `credential_mode`: **(string)** controls how the ue4-source Dockerfile securely obtains credentials for authenticating with remote git repositories when `source_mode` is set to `git`. Valid options are:
- `endpoint`: the default mode, whereby ue4-docker exposes an HTTP endpoint that responds with credentials when presented with a randomly-generated security token, which is injected into the ue4-source container during the build process by way of a Docker build argument. This mode will not work when exporting generated Dockerfiles, since the credential endpoint will not be available during the build process.
- `secrets`: **(Linux containers only)** uses BuildKit build secrets to securely inject the git credentials into the ue4-source container during the build process.
- `buildgraph_args`: **(string)** allows you to specify additional arguments to pass to the BuildGraph system when creating an Installed Build of the Unreal Engine in the ue4-minimal image.
- `disable_labels`: **(boolean)** prevents ue4-docker from applying labels to built container images. This includes the labels which specify the components excluded from the ue4-minimal image as well as the sentinel labels that the ue4-docker clean command uses to identify container images, and will therefore break the functionality of that command.
- `disable_all_patches`: **(boolean)** disables all the patches that ue4-docker ordinarily applies to the Unreal Engine source code. This is useful when building a custom fork of the Unreal Engine to which the appropriate patches have already been applied, **but will break the build process when used with a version of the Unreal Engine that requires one or more patches**. It is typically safer to disable individual patches using the specific flag for each patch instead of simply disabling everything:
- `disable_release_patches`: **(boolean)** disables the patches that ue4-docker ordinarily applies to versions of the Unreal Engine which are known to contain bugs, such as Unreal Engine 4.25.4. This will obviously break the build process when building these known broken releases, but will have no effect when building other versions of the Unreal Engine.
- `disable_windows_setup_patch`: **(boolean)** prevents ue4-docker from patching `Setup.bat` under Windows to comment out the calls to the Unreal Engine prerequisites installer and UnrealVersionSelector, both of which are known to cause issues during the build process for Windows containers.
- `disable_linker_fixup`: **(boolean)** prevents ue4-docker from replacing the linker in the Unreal Engine's bundled toolchain with a symbolic link to the system linker under Linux.
- `disable_example_platform_cleanup`: **(boolean)** prevents ue4-docker from removing the `Engine/Platforms/XXX` directory that was introduced in Unreal Engine 4.24.0 and subsequently removed in Unreal Engine 4.26.0. This directory represents a "dummy" target platform for

demonstration purposes, and the presence of this directory will typically break the build process.

- `disable_ubt_patches`: **(boolean)** disables the patches that ue4-docker ordinarily applies to fix bugs in UnrealBuildTool (UBT) under various versions of the Unreal Engine.
- `disable_opengl_patch`: **(boolean)** prevents ue4-docker from attempting to re-enable the OpenGL RHI under Linux for versions of the Unreal Engine in which it is present but deprecated.
- `disable_buildgraph_patches`: **(boolean)** disables the patches that ue4-docker ordinarily applies to the BuildGraph XML files used to create an Installed Build of the Unreal Engine. These patches fix various bugs under both Windows and Linux across multiple versions of the Unreal Engine.
- `disable_target_patches`: **(boolean)** disables the patches that ue4-docker ordinarily applies to fix broken `PlatformType` fields for client and server targets in `BaseEngine.ini` under Unreal Engine versions where these values are set incorrectly.
- `disable_unrealpak_copy`: **(boolean)** prevents ue4-docker from ensuring the UnrealPak tool is correctly copied into Installed Builds of the Unreal Engine under Linux. Some older versions of the Unreal Engine did not copy this correctly, breaking the functionality of created Installed Builds.
- `disable_toolchain_copy`: **(boolean)** prevents ue4-docker from ensuring the bundled clang toolchain is correctly copied into Installed Builds of the Unreal Engine under Linux. Some older versions of the Unreal Engine did not copy this correctly, breaking the functionality of created Installed Builds.

Windows-specific options

Specifying the Windows Server Core base image tag

The `-basetag` flag controls how the ue4-build-prerequisites image is built and tagged, which has a flow-on effect to all the other images. If you are building multiple related images over separate invocations of the build command (e.g. building the ue4-source image in one command and then subsequently building the ue4-minimal image in another command), be sure to specify the same `-basetag` flag each time to avoid unintentionally building two sets of unrelated images with different configurations.

By default, Windows container images are based on the Windows Server Core release that best matches the version of the host operating system. However, Windows containers cannot run a newer kernel version than that of the host operating system, rendering the latest images unusable under older versions of Windows 10 and Windows Server. (See the Windows Container Version Compatibility page for a table detailing which configurations are supported.)

If you are building images with the intention of subsequently running them under an older version of Windows 10 or Windows Server, you will need to build images based on the same kernel version as the target system (or older.) The kernel version can be specified by providing the appropriate base OS image tag via the `-basetag=TAG` flag when invoking the build command:

```
ue4-docker build 4.20.3 -basetag=ltsc2016 # Uses Windows Server 2016 (Long Term
Support Channel)
```

For a list of supported base image tags, see the Windows Server Core base image on Docker Hub.

Specifying the isolation mode under Windows

The isolation mode can be explicitly specified via the `-isolation=MODE` flag when invoking the build command. Valid values are `process` (supported under Windows Server and Windows 10 version 1809 or newer) or `hyperv` (supported under both Windows 10 and Windows Server.) If you do not explicitly specify an isolation mode then the appropriate default for the host system will be used.

Specifying Visual Studio Build Tools version under Windows

Keeping or excluding Installed Build debug symbols under Windows

Excluding debug symbols is necessary under some versions of Docker as a workaround for a bug that limits the amount of data that a `COPY` directive can process to 8GB. See this section of the Troubleshooting Build Issues page for further details on this issue.

Prior to version 0.0.30, ue4-docker defaulted to truncating all `.pdb` files when building the Installed Build for the ue4-minimal Windows image. This was done primarily to address the bug described in the warning alert above, and also had the benefit of reducing the overall size of the built container images. However, if you required the debug symbols for producing debuggable builds, you had to opt to retain all `.pdb` files by specifying the `--keep-debug` flag when invoking the build command. (This flag was removed in ue4-docker version 0.0.30, when the default behaviour was changed and replaced with a more generic, cross-platform approach.)

Since ue4-docker version 0.0.30, debug symbols are kept intact by default, and can be removed by using the `--exclude debug` flag as described in the section Excluding Engine components to reduce the final image size.

Building Linux container images under Windows

By default, Windows container images are built when running the build command under Windows. To build Linux container images instead, simply specify the `--linux` flag when invoking the build command.

Linux-specific options

Enabling CUDA support for GPU-enabled Linux images

The `--cuda` flag controls how the ue4-build-prerequisites image is built and tagged, which has a flow-on effect to all the other images. If you are building multiple related images over separate invocations of the build command (e.g. building the ue4-source image in one command and then subsequently building the ue4-minimal image in another command), be sure to specify the same `--cuda` flag each

> time to avoid unintentionally building two sets of unrelated images with different configurations.

By default, the Linux images built by ue4-docker support hardware-accelerated OpenGL when run via the NVIDIA Container Toolkit. If you would like CUDA support in addition to OpenGL support, simply specify the `--cuda` flag when invoking the build command.

You can also control the version of the CUDA base image that is used by appending a version number when specifying the `--cuda` flag, as demonstrated below:

```
# Uses the default CUDA base image (currently CUDA 9.2)
ue4-docker build RELEASE --cuda
```

```
# Uses the CUDA 10.0 base image
ue4-docker build RELEASE --cuda=10.0
{% endhighlight %}
```

For a list of supported CUDA versions, see the list of Ubuntu 18.04 image tags for the nvidia/cudagl base image.

Troubleshooting build issues

General issues

Building the `ue4-build-prerequisites` image fails with a network-related error

This indicates an underlying network or proxy server issue outside ue4-docker itself that you will need to troubleshoot. You can use the ue4-docker diagnostics command to test container network connectivity during the troubleshooting process. Here are some steps to try:

- If your host system accesses the network through a proxy server, make sure that Docker is configured to use the correct proxy server settings.
- If DNS resolution is failing then try adding the following entry to your Docker daemon configuration file (accessed through the Docker Engine settings pane if you're using Docker Desktop) and restarting the daemon:

```
{
  "dns": ["8.8.8.8"]
}
```

Cloning the UnrealEngine Git repository fails with the message `error: unable to read askpass response from 'C:\git-credential-helper.bat'` (for Windows containers) or `'/tmp/git-credential-helper.sh'` (for Linux containers)

This typically indicates that the firewall on the host system is blocking connections from the Docker

container, preventing it from retrieving the Git credentials supplied by the build command. This is particularly noticeable under a clean installation of Windows Server, which blocks connections from other subnets by default. The firewall will need to be configured appropriately to allow the connection, or else temporarily disabled. (Under Windows Server, the ue4-docker setup command can configure the firewall rule for you automatically.)

Building the Derived Data Cache (DDC) for the Installed Build of the Engine fails with a message about failed shader compilation or being unable to open a `.uasset` file

This is typically caused by insufficient available disk space. To fix this, simply free up some disk space and run the build again. Running docker system prune can be helpful for freeing up space occupied by untagged images. Note that restarting the Docker daemon and/or rebooting the host system may also help, since some versions of Docker have a bug that results in the amount of required disk space slowly increasing as more and more builds are run.

Building Windows containers fails with the message `hcsshim::ImportLayer failed in Win32: The system cannot find the path specified` or building Linux containers fails with a message about insufficient disk space

Assuming you haven't actually run out of disk space, this means that the maximum Docker image size has not been configured correctly.

- For Windows containers, follow the instructions provided by Microsoft, making sure you restart the Docker daemon after you've modified the config JSON. (Under Windows Server, the ue4-docker setup command can configure this for you automatically.)
- For Linux containers, use the Docker for Windows "Advanced" settings tab under Windows or the Docker for Mac "Disk" settings tab under macOS.

Building the `ue4-minimal` image fails on the `COPY --from=builder` directive that copies the Installed Build from the intermediate image into the final image

Modern versions of Docker Desktop for Windows and Docker EE for Windows Server suffer from issues with 8GiB filesystem layers, albeit due to different underlying bugs. Since ue4-docker version 0.0.47, you can use the ue4-docker diagnostics command to check whether the Docker daemon on your system suffers from this issue. If it does, you may need to exclude debug symbols when building Windows images.

Some versions of Docker contain one or more of a series of separate but related bugs that prevent the creation of filesystem layers which are 8GiB in size or larger:

- https://github.com/moby/moby/issues/37581 (affects all platforms)
- https://github.com/moby/moby/issues/40444 (affects Windows containers only)

#37581 was fixed in Docker CE 18.09.0, whilst #40444 was fixed in Docker CE 20.10.0.

If you are using a version of Docker that contains one of these bugs then you will need to exclude debug symbols, which reduces the size of the Installed Build below the 8GiB threshold. If debug symbols are required then it will be necessary to upgrade or downgrade to a version of Docker that

does not suffer from the 8GiB size limit issue (although finding such a version under Windows may prove quite difficult.)

Linux-specific issues

Building the Engine in a Linux container fails with an error indicating that a compatible version of clang cannot be found or the file `ToolchainVersion.txt` is missing

This is typically caused by the download of the Unreal Engine's toolchain archive from the Epic Games CDN becoming corrupted and failing to extract correctly. This issue can occur both inside containers and when running directly on a host system, and the fix is to simply delete the corrupted files and try again:

1. Untag the available-container-images.adoc#ue4-source[ue4-source] image.
2. Clear the Docker filesystem layer cache by running docker system prune.
3. Re-run the ue4-docker build command.

Windows-specific issues

Building the `ue4-build-prerequisites` image fails with an unknown error

Microsoft issued a security update in February 2020 that broke container compatibility for all versions of Windows Server and caused 32-bit applications to fail silently when run. The issue is resolved by ensuring that both the host system and the container image are using versions of Windows that incorporate the fix:

- Make sure your host system is up-to-date and all available Windows updates are installed.
- Make sure you are using the latest version of ue4-docker, which automatically uses container images that incorporate the fix.

Building Windows containers fails with the message `hcsshim: timeout waiting for notification extra info` or the message `This operation ended because the timeout has expired`

Recent versions of Docker under Windows may sometimes encounter the error hcsshim: timeout waiting for notification extra info when building or running Windows containers. This is a known issue when using Windows containers in Hyper-V isolation mode. At the time of writing, Microsoft have stated that they are aware of the problem, but an official fix is yet to be released.

As a workaround until a proper fix is issued, it seems that altering the memory limit for containers between subsequent invocations of the `docker` command can reduce the frequency with which this error occurs. (Changing the memory limit when using Hyper-V isolation likely forces Docker to provision a new Hyper-V VM, preventing it from re-using an existing one that has become unresponsive.) Please note that this workaround has been devised based on my own testing under Windows 10 and may not hold true when using Hyper-V isolation under Windows Server.

To enable the workaround, specify the `--random-memory` flag when invoking the build command. This will set the container memory limit to a random value between 10GB and 12GB when the build command starts. If a build fails with the `hcsshim` timeout error, simply re-run the build command

and in most cases the build will continue successfully, even if only for a short while. Restarting the Docker daemon may also help.

Note that some older versions of UnrealBuildTool will crash with an error stating **"The process cannot access the file because it is being used by another process"** when using a memory limit that is not a multiple of 4GB. If this happens, simply run the build command again with an appropriate memory limit (e.g. `-m 8GB` or `-m 12GB`.) If the access error occurs even when using an appropriate memory limit, this likely indicates that Windows is unable to allocate the full amount of memory to the container. Rebooting the host system may help to alleviate this issue.

Building or running Windows containers fails with the message `The operating system of the container does not match the operating system of the host`

This error is shown in two situations:

- The host system is running an **older kernel version** than the container image. In this case, you will need to build the images using the same kernel version as the host system or older. See Specifying the Windows Server Core base image tag for details on specifying the correct kernel version when building Windows container images.
- The host system is running a **newer kernel version** than the container image, and you are attempting to use process isolation mode instead of Hyper-V isolation mode. (Process isolation mode is the default under Windows Server.) In this case, you will need to use Hyper-V isolation mode instead. See Specifying the isolation mode under Windows for details on how to do this.

Pulling the .NET Framework base image fails with the message `ProcessUtilityVMImage \\?\`(long path here)`\UtilityVM: The system cannot find the path specified`

This is a known issue when the host system is running an older kernel version than the container image. Just like in the case of **"The operating system of the container does not match the operating system of the host"** error mentioned above, you will need to build the images using the same kernel version as the host system or older. See Specifying the Windows Server Core base image tag for details on specifying the correct kernel version when building Windows container images.

Building the Engine in a Windows container fails with the message `The process cannot access the file because it is being used by another process`

This is a known bug in some older versions of UnrealBuildTool when using a memory limit that is not a multiple of 4GB. To alleviate this issue, specify an appropriate memory limit override (e.g. `-m 8GB` or `-m 12GB`.) For more details on this issue, see the last paragraph of the `hcsshim` timeout issues section.

Building the Engine in a Windows container fails with the message `fatal error LNK1318: Unexpected PDB error; OK (0)`

This is a known bug in some versions of Visual Studio which only appears to occur intermittently. The simplest fix is to simply reboot the host system and then re-run the build command. Insufficient available memory may also contribute to triggering this bug. Note that a linker wrapper was added in Unreal Engine 4.24.0 to automatically retry link operations in the event that this bug occurs, so it shouldn't be an issue when building version 4.24.0 or newer.

Building the Engine in a Windows container fails with the message `fatal error C1060: the compiler is out of heap space`

This error typically occurs when the Windows pagefile size is not large enough. As stated in the Troubleshooting build issues, there is currently no exposed mechanism to control the pagefile size for containers running in Hyper-V isolation mode. However, containers running in process isolation mode will use the pagefile settings of the host system. When using process isolation mode, this error can be resolved by increasing the pagefile size on the host system. (Note that the host system will usually need to be rebooted for the updated pagefile settings to take effect.)

Building an Unreal project in a Windows container fails when the project files are located in a directory that is bind-mounted from the host operating system

The paths associated with Windows bind-mounted directories inside Hyper-V isolation mode VMs can cause issues for certain build tools, including UnrealBuildTool and CMake. As a result, building Unreal projects located in Windows bind-mounted directories is not advised when using Hyper-V isolation mode. The solution is to copy the Unreal project to a temporary directory within the container's filesystem and build it there, copying any produced build artifacts back to the host system via the bind-mounted directory as necessary.

Command reference

ue4-docker-build (1)

Name

ue4-docker-build - build container image for a specific version of Unreal Engine

Synopsis

ue4-docker build [*option*]... *version*

Description

To build container images for a specific version of the Unreal Engine, simply specify the *version* that you would like to build using full semver syntax. For example, to build Unreal Engine 4.20.3, run:

```
ue4-docker build 4.20.3
```

You will be prompted for the Git credentials to be used when cloning the Unreal Engine GitHub repository (this will be the GitHub username and password you normally use when cloning https://github.com/EpicGames/UnrealEngine. The build process will then start automatically, displaying progress output from each of the `docker build` commands that are being run in turn.

By default, all available images will be built. See the List of available container images for details on customising which images are built.

Options

-basetag *basetag*

Operating system base image tag to use. For Linux this is the version of Ubuntu (default is ubuntu18.04). For Windows this is the Windows Server Core base image tag (default is the host OS version)

-branch *branch*

Set the custom branch/tag to clone when **custom** is specified as the *version*.

--combine

Combine generated Dockerfiles into a single multi-stage build Dockerfile

-conan-ue4cli *conan_ue4cli*

Override the default version of conan-ue4cli installed in the ue4-full image

--dry-run

Use this if you would like to see what Docker commands would be run by `ue4-docker build`

without actually building anything. Execution will proceed as normal, but no Git credentials will be requested and all Docker commands will be printed to standard output instead of being executed as child processes.

--exclude {ddc,debug,templates}

Exclude the specified component from the ue4-minimal and ue4-full images.

The following components can be excluded:

- `ddc`: disables building the DDC for the Engine. This significantly speeds up building the Engine itself but results in far longer cook times when subsequently packaging Unreal projects.
- `debug`: removes all debug symbols from the built images.
- `templates`: removes the template projects and samples that ship with the Engine.

You can specify the `--exclude` flag multiple times to exclude as many components as you like. For example:

```
# Excludes both debug symbols and template projects
ue4-docker build 4.21.2 --exclude debug --exclude templates
```

-h, --help

Print help and exit

-interval *inverval*

Sampling interval in seconds when resource monitoring has been enabled using --monitor (default is 20 seconds)

-layout *layout*

Copy generated Dockerfiles to the specified directory and don't build the images

-m *memory*

Override the default memory limit under Windows (also overrides --random-memory)

--monitor

Monitor resource usage during builds (useful for debugging)

--no-cache

Disable Docker build cache

--no-full

Don't build the ue4-full image (deprecated, use **--target** *target* instead)

--no-minimal

Don't build the ue4-minimal image (deprecated, use **--target** *target* instead)

--opt *opt*

Set an advanced configuration option (can be specified multiple times to specify multiple options)

-password *password*

Specify access token or password to use when cloning the git repository

--rebuild

Rebuild images even if they already exist

-repo *repo*

Set the URL of custom git repository to clone when **custom** is specified as the *version*

-suffix *suffix*

Add a suffix to the tags of the built images

--target *target*

Tells ue4-docker to build specific image (including its dependencies).

Supported values: `all`, `build-prerequisites`, `full`, `minimal`, `source`.

You can specify the `--target` option multiple times.

-ue4cli *ue4cli*

Override the default version of ue4cli installed in the ue4-full image

-username *username*

Specify the username to use when cloning the git repository

-v, --verbose

Enable verbose output during builds (useful for debugging)

Linux-specific options

--cuda *version*

Add CUDA support as well as OpenGL support

Windows-specific options

--ignore-blacklist

Run builds even on blacklisted versions of Windows (advanced use only)

-isolation {process,hyperv}

Set the isolation mode to use

--linux

Use Linux containers under Windows hosts (useful when testing Docker Desktop or LCOW support)

--random-memory

Use a random memory limit for Windows containers

--visual-studio {2017,2019,2022}

Specify Visual Studio Build Tools version.

By default, ue4-docker uses Visual Studio Build Tools 2017 to build Unreal Engine. Starting with Unreal Engine 4.25, you may choose to use Visual Studio Build Tools 2019 instead.

Unreal Engine 5.0 adds support for VS2022 but removes support for VS2017.

Environment

This section describes several environment variables that affect how `ue4-docker build` operates.

UE4DOCKER_TAG_NAMESPACE

If you would like to override the default `adamrehn/` prefix that is used when generating the tags for all built images, you can do so by specifying a custom value using the `UE4DOCKER_TAG_NAMESPACE` environment variable.

See also

ue4-docker-clean(1)

ue4-docker-clean (1)

Name

ue4-docker-clean - cleans built container images.

Synopsis

ue4-docker clean [**-tag** *tag*] [**--source**] [**--all**] [**--dry-run**]

Description

By default, only dangling intermediate images leftover from ue4-docker multi-stage builds are removed.

Options

--all

Remove all ue4-docker images, applying the tag filter if one was specified

--dry-run

If you're unsure as to exactly what images will be removed by a given invocation of the command, append the `--dry-run` flag to have ue4-docker print the generated `docker rmi` commands instead of running them.

--prune

Run `docker system prune` after cleaning

--source

Remove ../building-images/available-container-images.adoc#ue4-source[ue4-source] images, applying the tag filter if one was specified

-tag *tag*

Apply a filter for the three flags below, restricting them to removing only images with the specified *tag* (e.g. `-tag 4.21.0` will only remove images for 4.21.0)

ue4-docker-diagnostics (1)

Name

ue4-docker-diagnostics - run diagnostics to detect issues with the host system configuration.

Synopsis

ue4-docker diagnostics *diagnostic*

Description

This command can be used to run the following diagnostics:

Checking for the Docker 8GiB filesystem layer bug

Some versions of Docker contain one or more of a series of separate but related bugs that prevent the creation of filesystem layers which are 8GiB in size or larger. This also causes `COPY` directives to fail when copying data in excess of 8GiB in size, breaking Dockerfile steps during the creation of Installed Builds that contain debug symbols.

This diagnostic tests whether the host system's Docker daemon suffers from this issue, by attempting to build a simple test Dockerfile with an 8GiB filesystem layer:

```
ue4-docker diagnostics 8gig
```

Checking for container network connectivity issues

This diagnostic tests whether running containers are able to access the internet, resolve DNS entries, and download remote files:

```
ue4-docker diagnostics network
```

ue4-docker-export (1)

Name

ue4-docker-export - export components from built container image to the host system

Synopsis

ue4-docker export *component tag destination*

This command can be used to export the following components:

Description

Exporting Installed Builds under Linux

Installed Builds of Unreal Engine can be exported to the host system starting with version 4.21.0. Once you have built either the ue4-minimal or ue4-full image for the UE4 version that you want to export, you can export it to the host system like so:

```
# Example: specify a version without a full image tag (assumes `adamrehn/ue4-full`)
# Exports the Installed Build from `adamrehn/ue4-full:4.27.0` to the directory
`~/UnrealInstalled` on the host system
ue4-docker export installed "4.27.0" ~/UnrealInstalled
```

```
# Example: specify a full image tag
# Exports the Installed Build from `adamrehn/ue4-minimal:4.27.0` to the directory
`~/UnrealInstalled` on the host system
ue4-docker export installed "adamrehn/ue4-minimal:4.27.0" ~/UnrealInstalled
```

```
# Example: use a container image based on ue4-minimal with a completely different tag
# Exports the Installed Build from `ghcr.io/epicgames/unreal-engine:dev-4.27.0` to the
directory `~/UnrealInstalled` on the host system
ue4-docker export installed "ghcr.io/epicgames/unreal-engine:dev-4.27.0"
~/UnrealInstalled
```

Exporting Conan packages

The Conan wrapper packages generated by `conan-ue4cli` can be exported from the ue4-full image to the local Conan package cache on the host system like so:

```
ue4-docker export packages 4.27.0 cache
```

Conan will need to be installed on the host system for this to work. To use the exported packages for development on the host system, you will also need to generate the accompanying profile-wide

packages by running the command:

```
ue4 conan generate --profile-only
```

This will require both `ue4cli` and `conan-ue4cli` to be installed on the host system.

ue4-docker-info (1)

Name

ue4-docker-info - display information about the host system and Docker daemon

Synopsis

ue4-docker info

Description

The command will output the following information:

- The ue4-docker version
- The host OS version
- The Docker daemon version
- Whether the NVIDIA Container Toolkit is supported under the current host configuration
- The detected configuration value for the maximum image size for Windows containers
- The total amount of detected system memory
- The number of detected physical and logical CPUs

ue4-docker-setup (1)

Name

ue4-docker-setup - automatically configure the host system where possible

Synopsis

ue4-docker setup

Description

This command will automatically configure a Linux or Windows Server host system with the settings required in order to build and run containers produced by ue4-docker.

Under Linux:

- If an active firewall is detected then a firewall rule will be created to allow Docker containers to communicate with the host system, which is required during the build of the ue4-source image.

Under Windows Server:

- The Docker daemon will be configured to set the maximum image size for Windows containers to 400GB.
- A Windows Firewall rule will be created to allow Docker containers to communicate with the host system, which is required during the build of the ue4-source image.
- Under Windows Server Core version 1809 and newer, any required DLL files will be copied to the host system from the full Windows base image. Note that the full base image was only introduced in Windows Server version 1809, so this step will not be performed under older versions of Windows Server.

Under Windows 10 and macOS this command will print a message informing the user that automatic configuration is not supported under their platform and that they will need to configure the system manually.

ue4-docker-test (1)

Name

ue4-docker-test - run tests to verify the correctness of built container images

Synopsis

ue4-docker test *tag*

Description

This command runs a suite of tests to verify that built ue4-full container images are functioning correctly and can be used to build and package Unreal projects and plugins.

This command is primarily intended for use by developers who are contributing to the ue4-docker project itself.

ue4-docker-version (1)

Name

ue4-docker-version - display version information about ue4-docker

Synopsis

ue4-docker version

Description

Prints ue4-docker version on the standard output.