Ship It

©Copyright Digital Software/Johan 'Zarkow' Munkestam

Version: 1.0.0 (Unity 6.3 LTS, 6000.3 and newer; editor tool plus a small runtime assembly; any render pipeline)

Introduction

Ship It turns the release chores of a multiplayer game into build profiles and buttons: client and dedicated server builds for Windows, Linux and macOS, a build stamp you can read at runtime, a SteamPipe upload, a Dockerfile for the Linux server and a command line for CI. Player Settings are never modified; every build gets its targets, scenes and defines from its profile.

Verified for 1.0.0 on Unity 6.3 LTS (6000.3.10): Windows client and server, Linux client and server, macOS client (built, not run), a Linux server running in Docker, a SteamPipe upload, and command-line builds. The generated GitHub Actions workflow is written and reviewed, but has not been run by us; see the CI chapter.

Getting started

1. Open Window > Digital Software > Ship It (also Tools > Ship It > Open Window).

2. Press New profile. A profile asset is created under Assets\ShipIt\Profiles, and its settings appear on the right. Pick the Target first; the name follows the target until you type your own.

3. Press Build. The output folder opens from Open output folder; the log under the buttons shows what was built, where, and with which git hash.

4. For a server, press New Linux server profile (or set a server target), install the Dedicated Server Build Support module if the window asks for it, and build. Write Dockerfile then puts everything for a container next to the build.

5. For Steam, tick Steam in the profile, enter your app and depot ids, set the Steam user in Settings, and use the Ship row: Write Steam VDFs, then Upload to Steam.

Profiles and the settings asset (Assets\ShipIt\ShipItSettings.asset) belong to your project: commit them with your game. Package updates never overwrite them.

The window

Left: every build profile in the project, with its target. Right: the selected profile. Under the profile, a preview line shows the resolved output folder, the scene count and the git state, and any problem that would stop the build is listed in orange.

Profile buttons

New profile, New Linux server profile: create a profile in the profiles folder. Duplicate: copies the selected profile. Delete: deletes it after asking. Settings: selects the settings asset. Refresh: reloads the list. Changes to a profile are saved to disk at once, because the command line and CI read the files.

Build row

Build: builds the selected profile. Build all: builds every profile in list order and stops at the first failure. Open output folder: opens the selected profile's output folder. Update Tools menu: writes Assets\ShipIt\Editor\ShipItProfileMenu.g.cs with one Tools > Ship It item per profile; press it again after adding or renaming profiles.

Ship row

Write Steam VDFs, Upload to Steam, Download SteamCMD (Steam chapter), Write Dockerfile (Containers chapter), Write GitHub Actions (CI chapter).

Profile fields

Enabled

On by default. Switch a profile off to let it rest without deleting it: Build all, -shipit-all, the Tools > Ship It > Profiles menu and the GitHub Actions export skip it. It stays in the list, greyed and marked (off), keeps all its settings, and can still be built on its own with Build or by naming it with -shipit-profile.

Target

Windows client, Windows server, Linux client, Linux server or macOS client. Server targets use Unity 6's dedicated server build (StandaloneBuildSubtarget.Server): no window, no graphics, UNITY_SERVER defined by Unity. They need the Dedicated Server Build Support module for that platform, and Linux targets also need Linux Build Support (Mono); the window names the missing module. macOS builds need Mac Build Support.

Profile name

Shown in the window and the Tools > Ship It menu, used by the command line (-shipit-profile) and by the {profile} token. The asset file is renamed to follow it. Two profiles with the same name are reported as a problem, since the menu and the command line find profiles by name.

Scenes and scene list

Build Settings: the scenes enabled in File > Build Profiles, in that order. Explicit list: the scenes in the list below, for example a server that loads a different first scene than the client.

Extra defines

Scripting defines for this build only, passed through BuildPlayerOptions.extraScriptingDefines. Player Settings are never changed, so the editor keeps compiling as before. Letters, digits and underscores only.

Output pattern

The build folder, relative to the project root or absolute. Tokens: {product} (product name), {version} (Player Settings version), {target} (windows-client, windows-server, linux-client, linux-server, mac-client), {profile} (profile name), {date} (yyyyMMdd), {git} (short hash, with -dirty when there are uncommitted changes; nogit without git). Default: Builds/{product}-{target}-{version}. The output may not be a drive root, the project folder, or inside Assets, Library, Packages, ProjectSettings, UserSettings, Logs or Temp; the window says so as you type.

Executable name

Without extension; empty uses the product name. The extension follows the target: .exe, .x86_64 or .app.

Development build

Unity's development build, with debug symbols. The zip and the Steam depot still leave the symbols out.

Clean output folder

On by default. Empties the output folder before building, so files from an earlier build (development DLLs, debug symbols, content you removed) never end up in the zip or on Steam. As a guard it only empties a folder that holds an earlier Unity build (a *_Data folder,

a UnityPlayer file or an .app) or is empty; any other folder stops the build with a message, so a mistyped pattern can never delete your files.

Zip after build

Writes <output folder>.zip next to the folder, without the files that match the exclusion list (debug symbols and Unity's DoNotShip folders by default).

Write build info

Writes build-info.json into StreamingAssets: product, version, build GUID, git hash, branch and dirty flag, target, profile, build time, Unity version, development flag and the server frame rate. See Build stamp.

Server arguments

Arguments the run script and the Dockerfile pass to a server build, after -batchmode -nographics. Default -logFile - (log to the console).

Server frame rate

Server targets only. A dedicated server has no frame cap and keeps a CPU core busy even when idle; in our test an idle Windows server used a full core without the cap and near zero with it. Default 30; 0 leaves the frame rate alone. The value travels in build-info.json (keep Write build info on) and is applied before the first scene loads. Override it at launch with -shipit-fps N, for example -shipit-fps 60, or -shipit-fps 0 for no cap.

Steam fields

Steam: enables the Steam buttons for this profile. App id and depot id: from Steamworks. Branch: the branch to set live after upload; leave it empty to upload without setting anything live, which is the safe default. Description: shown in Steamworks under SteamPipe > Builds; tokens allowed. File exclusions: SteamPipe wildcard patterns kept out of the depot and the zip.

Container fields

Docker image name: used in the generated commands; empty uses <product>-server. Container ports: one per entry, for example 7777/udp.

Settings

Profiles folder: where new profiles are created; profiles anywhere under Assets are found. Default output pattern: used by new profiles. SteamCMD path: steamcmd.exe or

steamcmd.sh; Download SteamCMD fills it. Steam user: the account for uploads; the password is never stored. ContentBuilder folder: receives the VDF files and SteamPipe's output, default Tools/ContentBuilder. Use git: read hash, branch and dirty state; needs git on the PATH.

Build stamp

With Write build info on, every build carries StreamingAssets/build-info.json. Read it at runtime:

```
var info = DigitalSoftware.ShipIt.BuildInfo.Current;

Debug.Log(info.Summary);   // "MyGame 1.2.0 (a1b2c3d4e-dirty, windows-client, 2026-10-17)"
```

Current is never null: in the editor, or in a build without the file, it describes the open project. Put Summary on your about screen and in bug reports, and a report always tells you which commit it came from, including whether the build had uncommitted changes (gitDirty).

Steam

Ship It follows Valve's ContentBuilder layout. Write Steam VDFs writes scripts/app_build_<app>.vdf and scripts/depot_build_<depot>.vdf into the ContentBuilder folder, with the profile's output folder as the depot content root, plus an upload script (upload_<app>.bat on Windows, .sh elsewhere). Upload to Steam runs SteamCMD in its own visible console with +login <user> +run_app_build: type the password and the Steam Guard code there. Nothing is stored by Ship It.

Download SteamCMD fetches Valve's SteamCMD into Tools/steamcmd next to the project and fills the path in Settings; on first run it updates itself. On macOS and Linux the archive is saved for you to extract.

After the first login SteamCMD keeps a login token in Tools/steamcmd/config. Ship It writes a .gitignore into that folder and into the SteamPipe output folder, so neither reaches your repository. The VDF scripts hold no secrets and can be committed.

Without a branch the build appears under SteamPipe > Builds in Steamworks and nothing changes for players; set it live by hand, or name a branch in the profile. Build the profile before uploading: the upload sends whatever is in the output folder.

Containers

For Linux server profiles, Write Dockerfile writes four files into the build folder: Dockerfile (debian:bookworm-slim, a non-root user, the server as entrypoint with -batchmode -nographics and the profile's server arguments, the profile's ports exposed), run.sh (starts the server without Docker), .dockerignore and DEPLOY.md with the exact commands for this build:

```
docker build -t <image>:<version> -t <image>:latest .
```

```
docker run --rm -p 7777:7777/udp <image>:<version>
```

Docker must be installed on the machine that builds the image; any Linux host with Docker runs it. Verified: the image builds, the server runs headless as the non-root user, logs to the console, and stops cleanly with docker stop.

DEPLOY.md also has notes for Edgegap (push the image to a registry, create an application version) and Hathora (upload a tarball of the folder, create a build and deployment). These follow each provider's documentation at the time of writing and have not been tested by us; check the provider's current documentation.

Command line and CI

```
Unity -batchmode -quit -projectPath <project> -executeMethod
DigitalSoftware.ShipIt.Editor.ShipItCli.Build -shipit-profile "Linux server" -logFile
build.log
```

-shipit-profile <name>: build that profile; repeat it for several. -shipit-all: build every profile. -shipit-output <folder>: build each profile into <folder>/<profile name> instead of its output pattern, a path without the version that CI steps can rely on. Exit code 0 when everything built, 1 otherwise. On Windows, start Unity with Start-Process -Wait (PowerShell) or start /wait (cmd) to read the exit code.

Write GitHub Actions writes .github/workflows/shipit.yml: one job per profile with game-ci/unity-builder calling the command line with -shipit-output Builds/ci, the build uploaded as an artifact, and for Steam profiles a game-ci/steam-deploy step with your app id and depot id. It runs on a manual trigger and on tags starting with v. Secrets: UNITY_LICENSE (or UNITY_EMAIL, UNITY_PASSWORD, UNITY_SERIAL), and for uploads STEAM_USERNAME, STEAM_PASSWORD and STEAM_CONFIG_VDF; game-ci's documentation describes their format.

The workflow is a starting point. We have reviewed it but not run it, since that needs a Unity licence in GitHub secrets. game-ci's build images may not include the Dedicated Server

modules; if a server job fails on a missing module, build servers on your own runner or with the local command line.

Demo

Assets\DigitalSoftware\ShipIt\Demo holds ShipItDemo.unity and three profiles: Demo Windows client, Demo Windows server and Demo Linux server. Build the Windows server and the Windows client, start the server from a console (ShipIt.exe -batchmode -nographics -logFile -) and then the client: the client window shows "connected" and the server's welcome with both build stamps every two seconds. The node is a UDP server on port 7777 in server builds and a client everywhere else; -port N and -connect host:port change the defaults. Plain .NET sockets, no netcode package.

The demo profiles appear in the window, the Tools menu and Build all. Untick Enabled on them to put them to rest, or delete the Demo folder when you no longer need it.

Limitations

Standalone targets only (Windows, Linux, macOS); no consoles or mobile. macOS builds were verified to build, not run. Uploads to itch.io or Epic are not included. Steam achievements and stats, store page assets and cloud deploy API calls are out of scope. The git fields need git on the PATH.

Further support

Head over to https://community.digitalsoftware.se/ or the Discord at https://discord.gg/fyqtxZt