Ways to contribute

Stride is a non-profit, community-driven, free and open source project. There are no full-time developers dedicated solely to Stride's advancement; instead, the engine progresses through the voluntary contributions of both the core team and the broader community.

In order to thrive, Stride requires the help from other community members. There are various ways you can help:

Community activity 

To make Stride better, just use it and tell others about it in your blogs, videos, and events. Get involved in discussions on Discord and GitHub Discussion. Being a user and spreading the word is vital for our engine, as we don't have a big marketing budget and rely on the community to grow.

Make games 

The best way to promote Stride is by creating a cool demo or, even better, a full game. Having people see and play an actual game made with Stride is the most effective form of advertisement.

Donate 

We utilize Open Collective for fundraising. The funds collected are allocated towards bug bounties and compensating individuals contracted for paid work.

Submit bug reports 

Making Stride more stable greatly improves usability and user satisfaction. So if you encounter a bug during development, please contribute by reporting it on GitHub.

PR reviews 

Contributing to Pull Requests (PRs) is excellent as it enables active participation without local builds. Reviewing and offering feedback in this collaborative process enhances code quality and maintains project standards, fostering a sense of community and knowledge sharing.

Contribute code 

If you're passionate about C# and want to contribute by building features or fixing bugs in Stride, dive into the source code and get involved! Have a look at the GitHub issues label Good first issue or funded Open Collective projects

Contribute to Documentation

Enhance the official documentation and tutorials by expanding the manual or creating textual/video guides. Your contributions will greatly improve accessibility and understanding for users.

Contribute to Website 

Enhance the official Stride website. Is design more your thing, or do you have an interesting blog post? It will all help us spread the word of Stride.

Donating to Stride

In order to support our contributors or if we want to finance a specific feature, we collect donations from individuals as well as organisations.

Open Collective

We gather funding through a website called OpenCollective. This website displays where all the money is coming from and where it is going to: 100% transparency guaranteed.

Projects

Stride's Open Collective hosts different 'Projects' — think of them as funding goals for specific features or contributions. Each project typically has a related GitHub ticket for more details on what's required for its development. If you're interested in working on or contributing to a particular feature, you can let us know by either replying:

- In each projects related GitHub thread and mention @stride3d/stride-contributors
- In the Stride Contributors channel on Discord

Contribute to Stride engine

Here you can find various pages describing building the source locally for different systems. You can also find information about Stride's architecture.

Contribute code

Want to help out fixing bugs or making new features? Check out how you can do so.

Bug bounties

Here you can learn about the process on our bug bounty process.

Building on Windows

Building and running the Stride engine locally on Windows using Visual Studio or other IDEs.

Localization

Learn how manage translations for the engine.

Hot reloading shaders

Learn about hot reloading shaders.

Source debugging

Learn how to do source debugging.

Visual studio plugin

Learn about the Visual studio plugin for shader development.

Architecture 

Build pipeline

Explanation on the current build pipeline.

Build details

Details on the building process of the Stride engine.

Graphics API

Stride support different graphics APIs. Here you can read more about them.

Dependency graph

A graphical overview of Stride's Assemblies, NameSpaces and Core methods.

Asset introspection

How and why the Stride engine uses asset introspection.

Contribute Code

If you are a developer and you want to help building Stride even more awesome, than you can do so in various ways.

Check our issue tracker

If you are just getting started with Stride, issues marked with 'good first issue' can be a good entry point. Please take a look at our issue tracker for other issues.

We also have funded Open Collective Projects in case you want to earn a little extra. These are either Bug bounties

Notify users

Once you start working on an issue, leave a message on the appropriate issue or create one if none exists to:

- You can always check on Github or Discord if you need to get started somewhere or if you need a general sense of approaching an issue.
- Make sure that no one else is working on that same issue
- Lay out your plans and discuss it with collaborators and users to make sure it is properly architectured and would fit well in the project

Coding style

Please use and follow Stride's `.editorconfig` when making changes to files.

Submitting Changes

- Push your changes to a specific branch in your fork.
- Use that branch to create and fill out a pull request to the official repository.
- After creating that pull request and if it's your first time contributing a CLA assistant will ask you to sign the .NET Foundation Contribution License Agreement.

Bug bounties

If you are a developer with solid experience in C#, rendering techniques, or game development, we want to hire you! We have allocated funds from supporters on OpenCollective and will pay you for your work on certain issues.

You can find issues with bounties here.

If the issue you want to work on doesn't have a bounty associated to it, feel free to get in touch with us by creating a new issue or adding your message to an existing one, tagging us with `@stride3d`/`@stride-contributors` and sharing your email address or Discord handle. You can also do it directly through Discord by sending a message in `#github-pr-and-issues` with the `@Developer` tag.

If you are interested in tackling one of those issues:

- Reply in the thread and tag `@stride3d`/`@stride-contributors`
- We'll get back to you and reserve that issue to your name.
- You can then create a new pull request and we'll review it.
- Once merged in you will receive 60% of the bounty and the other 40% on the next official release of the engine.

Payment info

Stride uses the Open source collective as our Fiscal host which approves the payments. They process payouts twice weekly, once they have been approved by the admins of the Collective. They make payments via PayPal and Wise, and can only make payouts to countries served by these payment processors.

You can go to the specific bug bounty on Stride's Open Collective for payment:

Building the source to Stride engine

Prerequisites

1. **Latest** Git **with Large File Support** selected in the setup on the components dialog.
2. DotNet SDK 6.0
   - Run `dotnet --info` in a console or powershell window to see which versions you have installed
3. Visual Studio 2022 with the following workloads:
   - `.NET desktop development` with `.NET Framework 4.7.2 targeting pack`
   - `Desktop development with C++` with
     - `Windows 10 SDK (10.0.18362.0)` (it's currently enabled by default but it might change)
     - `MSVC v143 - VS2022 C++ x64/x86 build tools (v14.30)` or later version (should be enabled by default)
     - `C++/CLI support for v143 build tools (v14.30)` or later version **(not enabled by default)**
   - Optional (to target iOS/Android): `Mobile development with .NET` and `Android SDK setup (API level 27)` individual component, then in Visual Studio go to `Tools > Android > Android SDK Manager` and install `NDK` (version 19+) from `Tools` tab.
4. **FBX SDK 2019.0 VS2015**

Build Stride with Visual studio 2022

Here are the steps to build Stride with Visual Studio. If you do not have or want to use Visual Studio, see building with other IDEs

1. Open a command prompt, point it to a directory and clone Stride to it: `git clone https://github.com/stride3d/stride.git`
   - Note that when you use GitHub -> Code -> Download ZIP, this doesn't support Large File Support `lfs`, make sure you use the command above or that your git client does it for you
2. Open `<StrideDir>\build\Stride.sln` with Visual Studio 2022 and build `Stride.GameStudio` in the 60-Editor solution folder (it should be the default startup project) or run it from VS's toolbar.
   - Optionally, open and build `Stride.Android.sln`, `Stride.iOS.sln`, etc.

If building failed

- If you skipped one of the `Prerequisites` thinking that you already have the latest version, update to the latest anyway just to be sure.
- Visual Studio might have issues properly building if an anterior version is present alongside 2022. If you want to keep those version make sure that they are up to date and that you are building Stride through VS 2022.
- Your system's *PATH* should not contain older versions of MSBuild (ex: `...\Microsoft Visual Studio\2019\BuildTools\MSBuild\Current\Bin` should be removed)
- Some changes might require a system reboot, try that if you haven't yet.

- Make sure that Git, Git LFS and Visual Studio can access the internet.
- Close VS, clear the nuget cache (in your cmd `dotnet nuget locals all --clear`), delete the hidden `.vs` folder inside `\build` and the files inside `bin\packages`, kill any msbuild and other vs processes, build the whole solution then build and run GameStudio.

Do note that test solutions might fail but it should not prevent you from building `Stride.GameStudio`.

Other IDEs

You are not required to use Visual Studio to build the Stride engine with Visual Studio. You can also build entirely from command line or other IDE's such as Rider or Visual Studio Code

Localization

You can help us translate Stride, by updating existing translations and/or adding new language at https://hosted.weblate.org/projects/stride/

Translation are manually merged back from `weblate` branch to `master` branch.

Activate new language in Game Studio

Once a new language has been added on weblate, it needs to be activated in the Game Studio during build & startup.

Please check commit https://github.com/stride3d/stride/commit/c70f07f449 for an example on how to add a new language in Game Studio.

Hot Reloading Engine Shaders in Editor

GameStudio automatically reloads project shaders on every file change, it can also reload engine shaders but the files the engine is looking at to synchronize those changes are located inside of the nuget packages `C:\Users\[USERNAME]\.nuget\packages\stride.rendering\4.1.0.1-beta\stride\Assets\Shadows\ShadowMapCommon.sdsl` for example.

If you still can't find where it's looking for with a specific file you can put a conditional breakpoint on the directoryWatcher.Track line with an expression like `filePath.Contains("NameOfYourShader")` and your IDE will break whenever that file is tracked, you can then inspect the value for `filePath` in your IDE/debugger's locals and it'll contain the full path to that file.

Don't forget to apply back the changes you made to the files in the nuget package to the files in your repo.

Setting Up Source Debugging in VS

First, make sure source debugging external dependencies is enabled:

- Make sure "Debug Just My Code" is disabled, in Tools -> Options -> Debugging.

Stride builds the PDB files right into the normal `.nupkg` files. When debugging a public release, SourceLink should cause Visual Studio to download source files right from github when stepping into them.

Because of the way Visual Studio tracks down source files while stepping, one can't Goto-Definition for types in dependencies in VS Community. The workaround is to first step into the dependency to get the source loaded. Alternatively, one can pay for .NET Reflector, VSPro, or Resharper, which fix this in Visual Studio.

- Set symbol (.pdb) and source files in the debugger - Visual Studio | Microsoft Docs

One day it might be nice to support `.snupkg` or `.source.nupkg` files, so the base packages could be smaller. However, it's not a big deal.

- Creating SourceLens symbol packages
- Source Link and .NET libraries | Microsoft Docs
- How to Debug a .NET Core NuGet Package

Related Discussions

- https://github.com/stride3d/stride/discussions/1116

Visual Studio Plugin

The Stride Visual Studio Plugin adds:

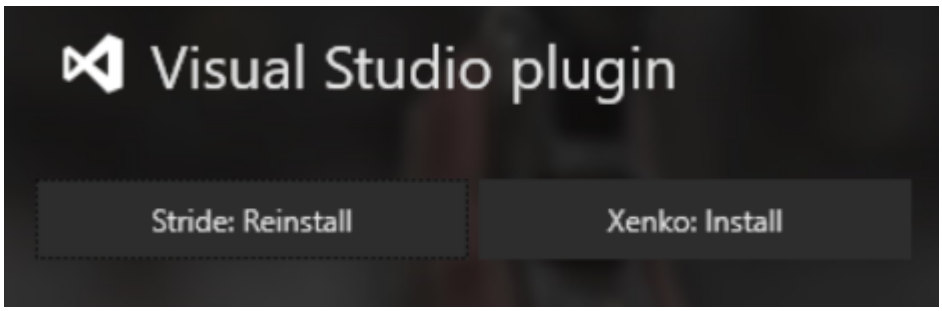
- syntax highlighting for Stride `.sdsl` shaders (formerly `.xlsl`)
- generates shader key files (`.sdsl.cs`) which create type definitions and ParameterKey values for the shader parameters
- generates shader effect files (`.sdfx.cs`)

Whenever you edit a shader file, the plug-in recompiles the C# key and effect files, so your C# code can reference elements necessary for your shader.

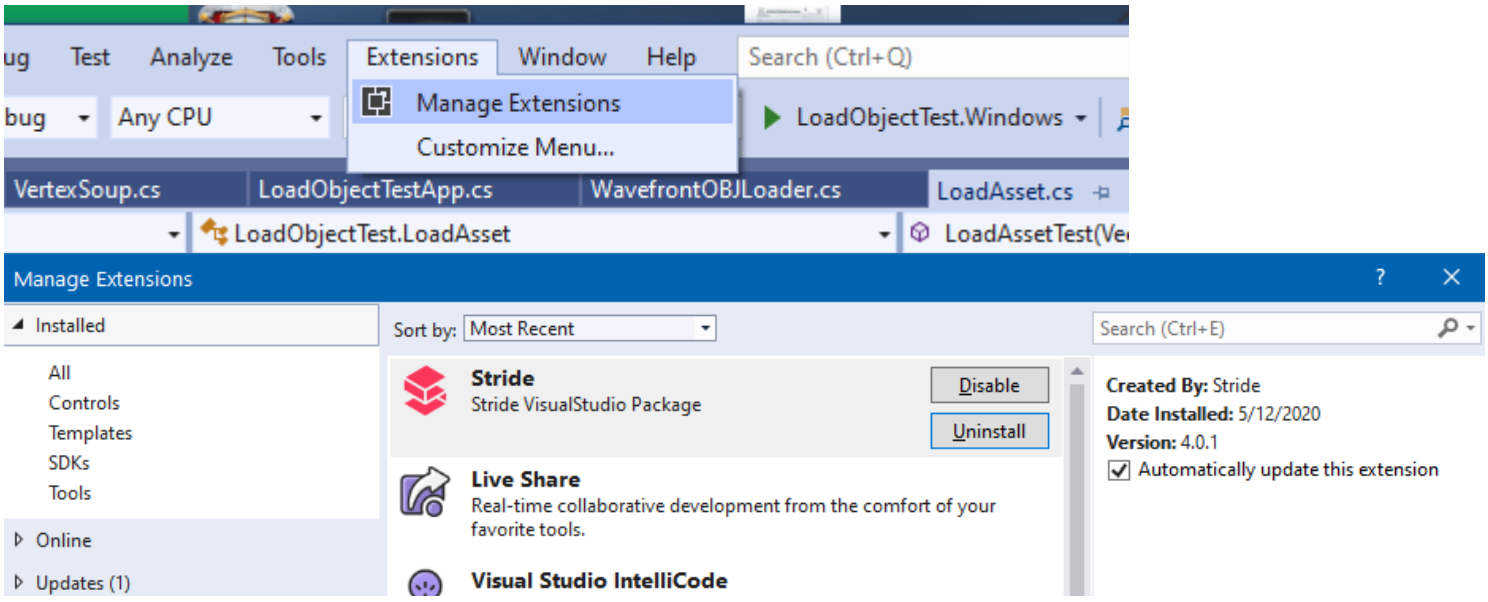
The code generation happens by looking at your game version loading the Shader Compiler dependency in the game build, and running the shader compiler.

The Visual Studio Plugin Code lives in:
https://github.com/stride3d/stride/tree/master/sources/tools/Stride.VisualStudio.Package

Install the Stride Visual Studio extension by using the button in the Stride Launcher:



You can check that the Stride Visual Studio plugin is installed in Visual Studio by going to Extensions-> Manage Extensions, and looking for the Stride Extension.



Engine architecture

General explanation of Stride engine source

Build pipeline

Explanation on the current build pipeline.

Build details

Details on the building process of the Stride engine.

Graphics API

Stride support different graphics APIs. Here you can read more about them.

Dependency graph

A graphical overview of Stride's Assemblies, NameSpaces and Core methods

Asset introspection

How and why the Stride engine uses asset introspection

Build pipeline

This document describes the Build pipeline in Stride, its current implementation (and legacy), and the work that should be done to improve it.

Terminology

- An **Asset** is a design-time object containing information to generate **Content** that can be loaded at runtime. For example, a **Model asset** contains the path to a source FBX file, and additional information such as an offset for the pivot point of the model, a scale factor, a list of materials to use for this model. A **Sprite font asset** contains a path to a source font, multiple parameters such as the size, kerning, etc. and information describing in which form it should be compiled (such as pre-rasterized, or using distance field...). **Asset** are serialized on disk using the YAML format, and are part of the data that a team developing a game should be sharing on a source control system.

- **Content** is the name given to compiled data (usually generated from **Asset**s) that can be loaded at runtime. This means that in term of format, **Content** is optimized for performance and size (using binary serialization, and data structured in a way so that the runtime can consume it without re-transforming it). Therefore **Content** is the platform-specific optimized version of your game data.

Design

Stride uses *Content-addressable storage* to store the data generated by the compilation. The main concept is that the actual name of each generated file is the hash of the file. So if, after a change, the resulting content built from the asset is different, then the file name will be different. An index map file contains the mapping between the content *URL* and the actual hash of the corresponding file. Parameters of each compilation commands are also hashed and stored in this database, so if a command is ran again with the same parameters, the build engine can easily recover the hashes of the corresponding generated files.

Build Engine

The build engine is the part of the infrastructure that transforms data from the **assets** into actual **content** and save it to the database. It was originally designed to build content from input similar to a makefile. (eg. "compile all files in `MyModels/*.fbx` into Stride models). It has then been changed to work with individual assets when the asset layer has been implemented. Due to this legacy, this library is still not perfectly suited or optimal to build assets in an efficient way (dependencies of build steps, management of a queue for live-compiling in the Game Studio, etc.).

Builder

The `Builder` class is the entry point of the build engine. A `Builder` will spawn a given number of threads, each one running a `Microthread` scheduler (see `RunUntilEnd` method).

Build Steps

The `Builder` takes a root `BuildStep` as input. We currently have two types of `BuildStep`s:

- A `ListBuildStep` contains a sequence of `BuildStep` (Formerly we had an additional parent class called `EnumerableBuildStep`, but it has been merged into `ListBuildStep`). A `ListBuildStep` will schedule all the build steps it contains at the same time, to be run in parallel. Formerly we had a synchronization mechanism using a special `WaitBuildStep` but it has been removed. We now use `PrerequisiteSteps` with `LinkBuildSteps` to manage dependencies.
- A `CommandBuildStep` contains a single `Command` to run, which does actual work to compile asset.

> **TODO:** Currently, when compiling a graph of build steps, we need to have all steps to compile in the root `ListBuildStep`. More especially, if we have a `ListBuildStep` container in which we want to put a step A that depends on a step B and C, we need to put A, B, C in the `ListBuildStep` container. This is cumbersome and error-prone. What we would like to do is to rely only on the `PrerequisiteSteps` of a given step to find what we have to compile. If we do so, we wouldn't need to return a `ListBuildStep` in `AssetCompilerResult`, but just the final build step for the asset, the graph of dependent build steps being described by recursive `PrerequisiteSteps`. The `ListBuildStep` container could be removed. We would still need to have lists of build steps when we compile multiple asset (eg. when compiling the full game), but it would be nothing that the build engine should be aware of.

Commands

Most command inherits from `IndexFileCommand`, which automatically register the output of the command into the command context.

Basically, at the beginning of the command (in the `PreCommand` method), a `BuildTransaction` object is created. This transaction contains a subset of the database of objects that have been already compiled, provided by the `ICommandContext.GetOutputObjectsGroups()`. In term of implementation, this method returns all the objects that where written by prerequisite build steps, and all the objects that are already written in any of the parent `ListBuildStep`s, recursively. The objects coming from the parent `ListBuildStep` are a legacy of when we were using `WaitBuildStep` to synchronize the build steps. This hopefully should be implemented differently, relying only on prerequisite (since no synchronization can happen in the `ListBuildStep itself, everything is run in parallel).

> **TODO:** Rewrite how OutputObjects are transfered from `BuildStep`s to other `BuildStep`s. Only the output from prerequisite `BuildStep` should be transfered. A lot of legacy makes this code very convoluted and hard to maintain.

The `BuildTransaction` created during this step is mounted as a *Microthread-local database*, which is accessible only from the current microthread (which is basically the current command).

At the end of the command (in the `PostCommand` method), every object that has been written in the database by the command are extracted from the `BuildTransaction` and registered to the current `ICommandContext` (which is how the `ICommandContext` can "flow" objects from one command to the other.

It's important to keep in mind that objects accessible in a given command (in the `DoCommandOverride`) using a `ContentManager` are those provided during the `PreCommand` step, and therefore it is important that dependencies between commands (what other commmands a command needs to be completed to start) are properly set.

Compilers

Compilers are classes that generate a set of `BuildStep`s to compile a given `Asset` in a specific context. This list could grow in the future if we have other needs, but the current different contexts are:

- compiling the asset for the game
- compiling the asset for the scene editor
- compiling the asset to display in the preview
- compiling the asset to generate a thumbnail

IAssetCompiler

This is the base interface for compiler. The entry point is the `Prepare` method, which takes an `AssetItem` and returns a `AssetCompilerResult`, which is a mix of a `LoggerResult` and a `ListBuildStep`. Usually there are two implementations per asset types, one to compile asset for the game and one to compile asset for its thumbnails. Some asset types such as animations might have an additional implementation for the preview.

Each implementation of `IAssetCompiler` must have the `AssetCompilerAttribute` attached to the class, in order to be registered (compilers are registered via the `AssetCompilerRegistry`.

> **TODO:** The `AssetCompilerRegistry` could be merged into the `AssetRegistry` to have a single location where asset-related types and meta-information are registered.

Each compiler provides a set of methods to help discover the dependencies between assets and compilers. They will be covered later in this document.

ICompilationContext

> Not to be mistaken with `CompilerContext` and `AssetCompilerContext`.

Contexts of compilation are defined by *types*, which allow to use inheritance mechanism to fallback on a default compiler when there is no specific compiler for a given context. Each compilation context type must implement `ICompilationContext`. Currently we have:

- `AssetCompilationContext` is the context used when we compile an asset for the runtime (ie. the game).
- `EditorGameCompilationContext` is the context used when we compile an asset for the scene editor, which is a specific runtime. Therefore, it inherits from `AssetCompilationContext`.
- `PreviewCompilationContext` is the context used when we compile an asset for the preview, which is a specific runtime. Therefore, it inherits from `AssetCompilationContext`.
- `ThumbnailCompilationContext` is the context used when we compile an asset to generate a thumbnail. Generally, for thumbnails, we compile one or several assets for the runtime, and use additional steps to generate the thumbnail with the `ThumbnailCompilationContext` (see below).

> **TODO:** Currently thumbnail compilation is in a poor state. In `ThumbnailListCompiler.Compile`, we first generate the steps to compile the asset in `PreviewCompilationContext`, then generate the steps to compile the asset in `ThumbnailCompilationContext`, and finally we like the first with the latter. Dependencies from thumbnail compilers (which load a scene and take screenshots) to the runtime compiler (which compile the asset) is **not** expressed at all. It just works now because in all current cases, the `PreviewCompilationContext` does what we need for thumbnails (for example, the `AnimationAssetPreviewCompiler` adds the preview model to the normal compilation of the animation, which is needed for both preview and thumbnail).

Dependency managers

We currently have two mechanisms that handle dependencies.

> **TODO:** Merge the `AssetDependencyManager` and the `BuildDependencyManager` together into a single dependency manager object. There is a lot of redundancy between both, one rely on the other, some code is duplicated. See `XK-4862`

AssetDependencyManager

The `AssetDependencyManager` was the first implementation of an mechanism to manage dependencies between assets. It works independently of the build, which is one of the main issue it had and the reason why we started to develop a new infrastructure.

It is based essentially on visiting assets with a `DataVisitorBase` to find references to other assets. There are two ways of referencing an asset:

- Having a property whose type is an implementation of `IReference`. More explicitely the only case we have currently is `AssetReference`. This type contains an `AssetId` and a `Location` corresponding to the referenced asset.
- Having a property whose type correspond to a *Content* type, ie. a type registered as being the compiled version of an asset type (for example, `Texture` is the Content type of `TextureAsset`).

The problem of that design was that once all the references are collected, there is no way to know of the referenced assets are actually consumed, which could be one of the three following way:

- the referenced asset is not needed to compile this asset, but it's needed at runtime to use the compiled content (eg. Models need Materials, who need Textures. But you can compile Models, Materials and Textures independently).
- the referenced asset needs to be compiled before this asset, and the compiler of this asset needs to load the corresponding content generated from the referenced asset (eg. A prefab model, which aggregates multiple models together, needs the compiled version of each model it's referencing to be able to merge them).
- the referenced asset is read when compiling this asset because it depends on some of its parameter, but the referenced asset itself doesn't need to be compiled first (eg. Navigation Meshes need to read the scene asset they are related to in order to gather static colliders it contains, but they don't need to compile the scene itself).

BuildDependencyManager

The `BuildDependencyManager` has been introduced recently to solve the problems of the `AssetDependencyManager`. It is currently not complete, and the ultimate goal is to merge it totally with the `AssetDependencyManager`.

The approach is a bit different. Rather than extracting dependencies from the asset itself, we extract them from the compilers of the assets, which are better suited to know what they exactly need to compile the asset and what will be needed to load the asset at runtime.

But one asset type can have multiple compilers associated to it (for the game, for the thumbnail, for the preview...). So the `BuildDependencyManager` works in the context of a specific compiler.

Currently there is one `BuildDependencyManager` for each type of compiler.

> **TODO:** Have a single global instance of `BuildDependencyManager` that contains all types of dependencies for all context of compilers. For example, we have thumbnail compilers that requires *game* version of assets, which means that the `BuildDependencyManager` for thumbnails will also contain a large part of the `BuildDependencyManager` to build the game. Merging everything into a single graph would reduce redundancy and risk to trigger the same operation multiple times simultaneously.

AssetDependenciesCompiler

The `AssetDependenciesCompiler` is the object that computes the dependencies with the `BuildDependencyManager`, and then generates the build steps for a given asset, including the runtime dependencies. It's the main entry point of compilation for the CompilerApp, the scene editor, and the preview. Thumbnails also use it, via the `ThumbnailListCompiler` class.

> **TODO:** This class should be removed, and its content moved into the `BuildDependencyManager` class. By doing so, it should be possible to make `BuildAssetNode` and `BuildAssetLink` internal - those classes are just the data of the dependency graph, they should not be exposed publicly. To do that, a method to retrieve the dependencies in a given context must be implemented in `BuildDependencyManager` in order to fix the usage of `BuildAssetNode` in `EditorContentLoader`.

In the Game Studio

The Game Studio compiles assets in various versions all the time. It has some specific way of managing database and content depending on the context.

Remark: the Game Studio never saves index file on the disk, it keeps the url -> hash mapping in memory, always.

Databases

Before accessing content to load, a Microthread-local database must be mounted. Depending on the context, it can be a database containing a scene and its dependencies (scene editor), the assets needed to create a thumbnail, an asset to display in the preview...

For the scene editor, this is handled by the `GameStudioDatabase` class. Thumbnails and preview also handle database mounting internally (in `ThumbnailGenerator` for example).

> **TODO:** See if it could be possible/useful to wrap all database-mounting in the Game Studio into the GameStudioDatabase class.

Builder service

All compilations that occur in the Game Studio is done through the `GameStudioBuilderService`. This class creates an instance of `Builder`, a `DynamicBuilder` which allows to feed the Builder with build steps at any time. Having a single builder for the whole Game Studio allows to control the number of threads and concurrent tasks more easily.

The `DynamicBuilder` class simply creates a thread to run the Builder on, and set a special build step, `DynamicBuildStep`, as root step of this builder. This step is permanently waiting for other child build step to be posted, and execute them.

> **TODO:** Currently the dynamic build step waits arbitratly with the `CompleteOneBuildStep` method when more than 8 assets compiling. This is a poor design because if the 8 assets are for example prefabs who contains a lot of models, materials, textures, it will block until all are done, although we could complete the thumbnails of these models/materials/textures individually. Ideally, this `await` should be removed, and a way to make sure thumbnails of assets which are compiled are created as soon as possible should be implemented.

The builder service uses `AssetBuildUnit`s as unit of compilation. A build unit corresponds to a single asset, and encapsulates the compiler and the generated build step of this asset.

EditorContentLoader

The scene editor needs a special behavior in term of asset loading. The main issue is that any type of asset can be modified by the user (for example a texture), and then need to be reloaded. Stride use the `ContentManager` to handle reference counting of loaded assets. With a few exception (Materials, maybe Textures), it does not support hot-swapping an asset. Therefore, when an asset needs to be reloaded, we actually need to unload and reload the *first-referencer* of this asset.

The *first-referencer* is the first asset referenced by an entity, that contains a way (in term of reference) to the asset to reload. For example, in case of a texture, we will have to reload all models that use materials that use the texture to reload.

This is done by the `EditorContentLoader` class. At initialization, this class collects all *first-referencer* assets and build them. Each time an asset is built, it is then loaded into the scene editor game, and the references (from the entity to the asset) are updated. This means that this class needs to track all first-referencers on its own and update them. This is done specifically by the `LoaderReferenceManager` object. The reference are collected from the `GameEditorChangePropagator`, an object that takes the responsibility to push synchronization of changes between the assets and the game (for all properties, including non-references). There is one instance of it per entity. When a property of an entity that contains a reference to an asset (a *first-referencer*) is modified, the propagator will trigger the work to compile and update the entity. In case of a referenced asset modified by the user, `EditorContentLoader.AssetPropertiesChanged` takes the responsibility to gather, build, unload and reload what needs to be reloaded.

Additional Todos

> **TODO:** `GetInputFiles` exists both in `Command` and in `IAssetCompiler`. It has the same signature in both case, so it's returning information using `ObjectUrl` and `UrlType` in the compiler, where we are trying to describe dependency. That signature should be changed, so it returns information using `BuildDependencyType` and `AssetCompilationContext`, just like the GetInputTypes method. Also, the method is passed to the command via the `InputFilesGetter` which is not very nice and has to be done manually (super error-prone, we had multiple commands that were missing it!). An automated way should be provided.

> **TODO:** The current design of the build steps and list build steps is a *tree*. For this reason, same build steps are often generated multiple times and appears in multiple trees. It could be possible to cache and share the build step if the structure was a *graph* rather than a *tree*. Do to that, the `Parent` property of build steps should be removed. The main difficulty is that the way output objects of build steps flow between steps has to be rewritten.

Build details

This is a technical description what happens in our build and how it is organized. This covers mostly the build architecture of Stride itself.

- Targets contains the MSBuild target files used by Games
- sources/common/targets (generic) and sources/targets (Stride-specific) contains the MSBuild target files used to build Stride itself.

Since 3.1, we switched from our custom build system to the new csproj system with one nuget package per assembly.

We use `TargetFrameworks` to properly compile the different platforms using a single project (Android, iOS, etc...).

Also, we use `RuntimeIdentifiers` to select graphics platform. MSBuild.Sdk.Extras is used to properly build NuGet packages with multiple `RuntimeIdentifiers` (not supported out of the box).

Limitations

- Dependencies are per `TargetFramework` and can't be done per `RuntimeIdentifier` (tracked in NuGet#1660).

NuGet resolver

Since we want to package tools (i.e. GameStudio, ConnectionRouter, CompilerApp) with a package that contains only the executable with proper dependencies to other NuGet runtime packages, we use NuGet API to resolve assemblies at runtime.

The code responsible for this is located in Stride.NuGetResolver.

Later, we might want to take advantage of .NET Core dependency resolving to do that natively. Also, we might want to use actual project information/dependencies to resolve to different runtime assemblies and better support plugins.

Versioning

Stride is versioned using `SharedAssemblyInfo.cs`. For example, assuming version `4.1.3.135+gfa0f5cc4`:

- `4.1` is the Stride major and minor version, as they are grouped in the launcher. Versions inside this group shouldn't have breaking changes
- `3` is the asset version. This can be bumped if asset files require some upgrade.
- `135` is the git height (number of commits since `4.1.3` is set), computed automatically when building packages. Note: when building packages locally, this will typically be 1. This is the reason why the

asset version needs to be bumped when asset changes to keep things ordered (otherwise the git height version `1` will always be lower than official version).
- `+gfa0f5cc4` means git commit `fa0f5cc4`

Assembly processor

Assembly processor is run by both Game and Stride targets.

It performs various transforms to the compiled assemblies:

- Generate DataSerializer serialization code (and merge it back in assembly using IL-Repack)
- Generate UpdateEngine code
- Scan for types or attributes with `[ScanAssembly]` to quickly enumerate them without needing `Assembly.GetTypes()`
- Optimize calls to Stride.Core.Utilities
- Automatically call methods tagged with ModuleInitializer
- Cache lambdas and various other code generation related to Dispatcher
- A few other internal tasks

For performance reasons, it is run as a MSBuild Task (avoid reload/JIT-ing). If you wish to make it run the executable directly, set `StrideAssemblyProcessorDev` to `true`.

Dependencies

We want an easy mechanism to attach some files to copy alongside a referenced .dll or .exe, including content and native libraries.

As a result, `<StrideContent>` and `<StrideNativeLib>` item types were added.

When a project declare them, they will be saved alongside the assembly with extension `.ssdeps`, to instruct referencing projects what needs to be copied.

Also, for the specific case of `<StrideNativeLib>`, we automatically copy them in appropriate folders and link them if necessary.

Note: we don't apply them transitively yet (project output won't contains the `.ssdeps` file anymore so it is mostly useful to reference from executables/apps directly)

Native

By adding a reference to `Stride.Native.targets`, it is easy to build some C/C++ files that will be compiled on all platforms and automatically added to the `.ssdeps` file.

Limitations

It seems that using those optimization don't work well with shadow copying and probing privatePath. This forces us to copy the `Direct3D11` specific assemblies to the top level `Windows` folder at startup of some tools. This is little bit unfortunate as it seems to disturb the MSBuild assembly searching (happens before `$(AssemblySearchPaths)`). As a result, inside Stride solution it is necessary to explicitly add `<ProjectReference>` to the graphics specific assemblies otherwise wrong ones might be picked up.

This will require further investigation to avoid this copying at all.

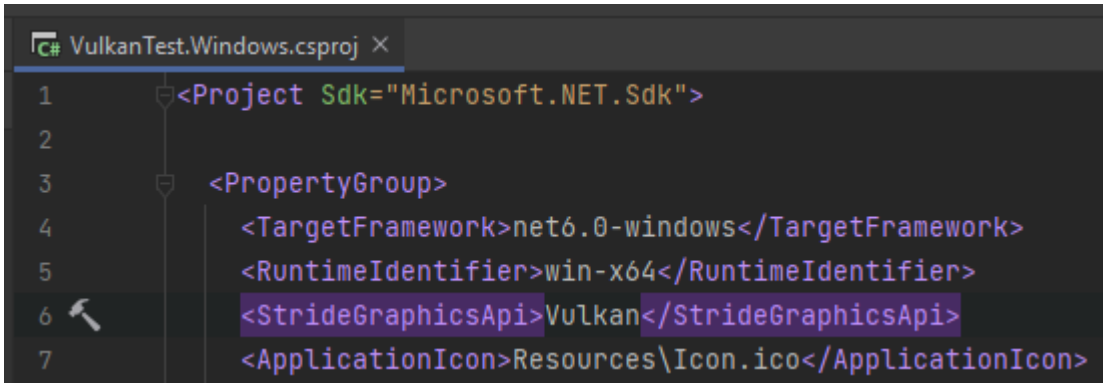
Asset Compiler

Both Games and Stride unit tests are running the asset compiler as part of the build process to create assets.

Graphics API

To run your projects through a different API than the default one, add the following line to the `PropertyGroup` of your executable's `.csproj` file:

```
<StrideGraphicsApi>Vulkan</StrideGraphicsApi>
```



Supported values are as follows:

```
Null
Direct3D11
Direct3D12
OpenGL
OpenGLES
Vulkan
```

You *may* also have to add `<PackageReference Include="Stride.Shaders.Compiler" Version="x.x.x.x" />` to your *main* `.csproj`, and don't forget to replace `Version` appropriately.

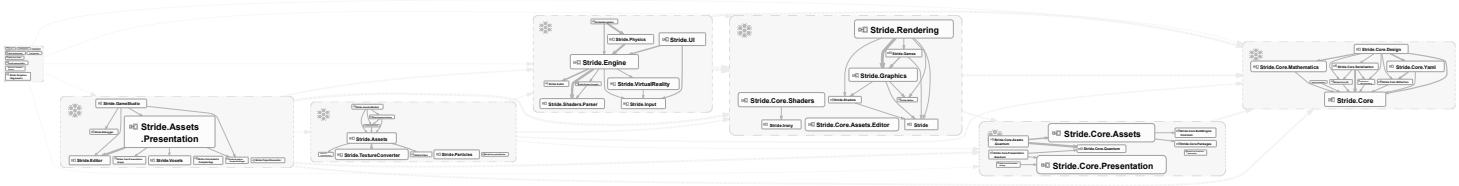
Engine

If you are using a local build of the engine you should run the build again with the following command:

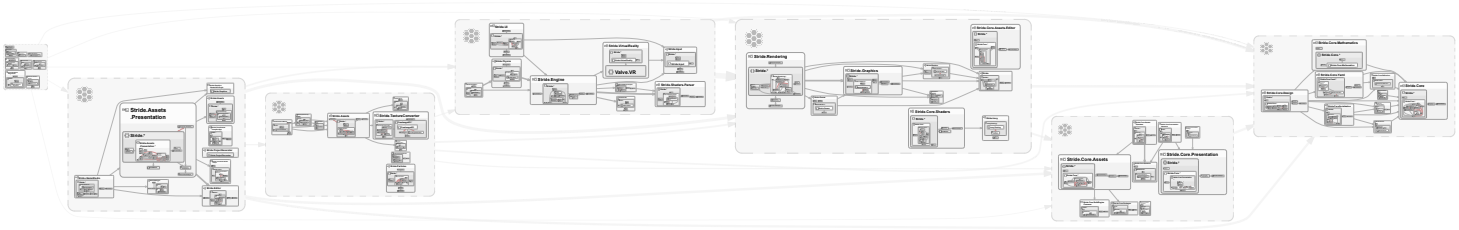
```
msbuild /t:Build /p:StrideGraphicsApiDependentBuildAll=true Stride.sln
```

Dependency graph

Assemblies



Namespaces

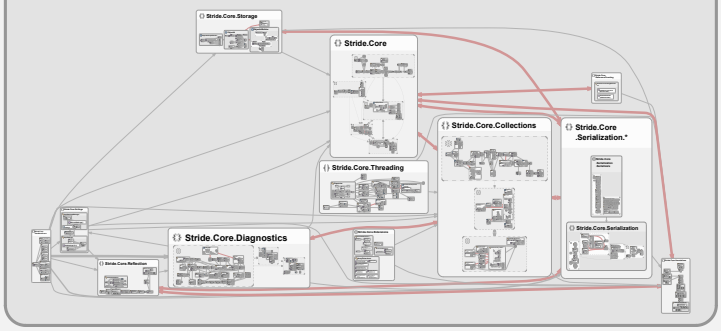


Stride.Core methods

Stride.Core

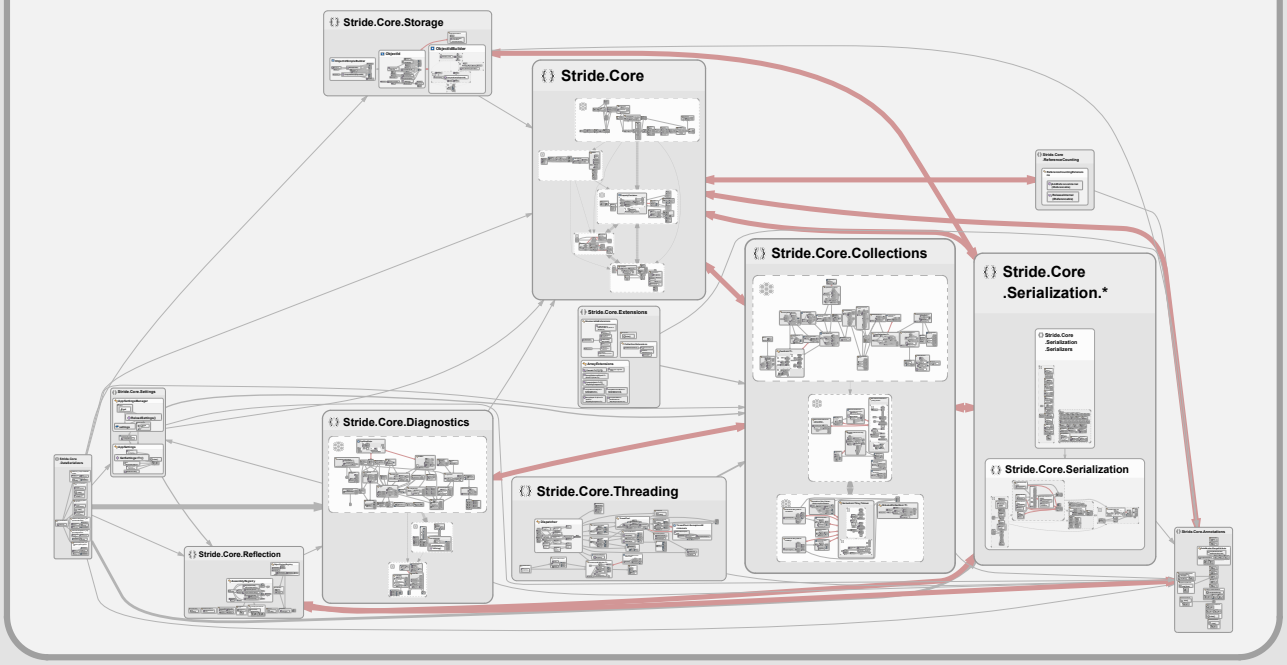
{ } Stride.*

{ } Stride.Core.*



Stride.Core

{ } Stride.Core.*



Asset, introspection and prefab

NOTE: Please read the Terminology section of the Build Pipeline documentation first

Design notes

Assets contains various properties describing how a given **Content** should be generated. Some constraints are defined by design:

- All types that can be referenced directly or indirectly by an asset must be serializable. This means that it should have the `[DataContract]` attribute, and the type of all its members must have it too.
- Members that cannot or should not be serialized can have the `[DataMemberIgnore]` attributes
- Other members can have additional metadata regarding serialization by using the `[DataMember]` attributes. There is also a large list of other attributes that can be used to customize serialization and presentation of those members.
- Arrays are not properly supported
- Any type of ordered collection is supported, but unordered collection (sets, bags) are not.
- Dictionaries are supported as long as the type of the key is a primitive type (see below for the definition of primitive type)
- When an asset references another asset, the member or item shouldn't use the type of the target asset, but the corresponding **Content**. For example, the `MaterialAsset` needs to reference a texture, it will have a `Texture` member and not a `TextureAsset`.
- It is possible to use the `AssetReference` type to represent a reference to any type of asset.
- Nullable value types are not properly supported
- An asset can reference multiple times the same objects through various members/items, but one of the member/item must be the "real instance", and the others must be defined as "object references", see below for more details.

Yaml metadata

When assets are serialized to/deserialized from Yaml files, dictionaries of metadata is created or consumed in the process. There is one dictionary per type of metadata. The dictionary maps a property path (using `YamlAssetPath`) to a value, and is stored in a instance of `YamlAssetMetadata`. These dictionary are exchanged between the low-level Yaml serialization layer and the asset-aware layer via the `AssetItem.Metadata` property. This property is not synchronized all the time, it is just consumed after deserialization, to apply metadata to the asset, and generated just before serialization, to allow the metadata to be consumed during serialization.

Overrides

The prefab and archetype system introduces the possibility to override properties of an asset. Some nodes of the property tree of an asset might have a *base*. (usually all of them in case of archetype, and some specific entities that are prefab instances in case of scene). How nodes are connected together is

explained later on this documentation, but from a serialization point of view, any property that is overridden will have associated yaml metadata. Then we usa a custom serializer backend, `AssetObjectSerializerBackend`, that will append a star symbol `*` at the end of the property name in Yaml.

Collections

Collections need special handling to properly support override. An item of a collection that is inherited from a base can be either modified (have another value) or deleted. Also, new items that are not present in the base can have been added. This is problematic in the case of ordered collection such as `List` because adding/deleting items changes the indices of item.

To solve all these issues, we introduce an object called `CollectionItemIdentifiers`. There is one instance of this object per collection that supports override. This instance is created or retrieved using the `CollectionItemIdHelper`. They are stored using `ShadowObject`, which maintain weak references from the collection to the `CollectionItemIdentifiers`. This means that it is currently not possible to have overridable items in collection that are `struct`.

A collection that can't or shouldn't have overridable items should have the `NonIdentifiableCollectionItemsAttribute`.

The `CollectionItemIdentifiers` associates an item of the collection to a unique id. It also keep track of deleted items, to be able to tell, when an item in an instance collection is missing comparing to the base collection, if it's because it has been removed purposely from the instance collection, or if it's because it has been added after the instance collection creation to the base collection.

Items, in the `CollectionItemIdentifiers`, are represented by their key (for dictionaries) or index (list). This means that any collection operation (add, remove...) must call the proper method of this class to properly update this collection. This is automatically done as long as the collection is updated through Quantum (see below).

In term of inheritance and override, the item id is what connect a given item of the base to a given item of the instance. This means that items can be re-ordered, and other items can be inserted, without loosing or messing the connection between base and instances. Also, for dictionary, keys can be renamed in the instance.

At serialization, the item id is written in front of each item (so collections are transformed to dictionaries of [`ItemId`, `TValue`] and dictionary are transformed to dictionaries of [`KeyWithId<TKey>`, `TValue`], with `KeyWithId` being equivalent to a Tuple). Here is an example of Yaml for a base collection and an instance collection:

Base collection, with one id per item:

```
Strings:
    309e0b5643c5a94caa799a5ea1480617: Hello
    e09ec493d05e0446b75358f0e1c0fbdd: World
    9550f04dcee1d24fa8a30e41eea71a94: Example
    1da8adce3f0ce9449a9ed0e48cd32f20: BaseClass
```

Derived collection. The first item is overridden, the 4th is a new item (added), and the last one express that the `BaseClass` entry has been deleted in the derived instance.

```
Strings:
    309e0b5643c5a94caa799a5ea1480617*: Hi
    e09ec493d05e0446b75358f0e1c0fbdd: World
    9550f04dcee1d24fa8a30e41eea71a94: Example
    cfce75d38d66e24fae426d1f40aa4f8a*: Override
    1da8adce3f0ce9449a9ed0e48cd32f20: ~(Deleted)
```

When two assets that are connected with a base relationship are loaded, it is then possible to reconcile them:

- any item missing in the derived collection is re-added (so the `~(Deleted)` is need to purposely delete items)
- any item existing in the derived collection that doesn't exist in the base collection and doesn't have the star `*` is removed
- any item that exists in both collection but have a different value is overwritten with the value of the base collection
- overridden items (with the star `*`) are untouched

Quantum

In Stride, we use an introspection framework called *Quantum*.

Type descriptors

The first layer used to introspect object is in `Stride.Core.Reflection`. This assembly contains type descriptors, which are basically objects abstracting the reflection infrastructure. It is currently using .NET reflection (`System.Reflection`) but could later be implemented in a more efficient way (using `Expression`, or IL code).

The `TypeDescriptorFactory` allows to retrieve introspection information on any type. `ObjectDescriptor`s contains descriptor for members which allow to access them. Collections, dictionaries and arrays are also handled (NOTE: arrays are not fully supported in Quantum itself).

This assembly also provides an `AttributeRegistry` which allows to attach `Attribute`s to any class or member externally.

> **TODO:** make sure all locations where we read `Attribute`s are using the `AttributeRegistry` and not the default .NET methods, so we properly support externally attached attributes.

Node graphs

In order to introspect object, we build graphs on top of each object, representing their members, and referencing the graphs of other objects they reference through members or collection. The classes handling theses graphs are in the `Stride.Core.Quantum` assembly.

Node containers

Nodes of the graphs are created into an instance of `NodeContainer`. Usually a single instance of `NodeContainer` is enough, but we have some scenarios where we use multiple ones: for example each instance of scene editor contains its own `NodeContainer` instance to build graphs of game-side objects, which are different from asset-side (ie. UI-side) objects, have a different lifespan, and require different metadata.

In the GameStudio, the `NodeContainer` class has two derivations: the `AssetNodeContainer` class, which expands the primitive types to add Stride-specific types (such as `Vector3`, `Matrix`, `Guid`...). This class is inherited to a `SessionNodeContainer`, which additionally allows plugin to register their own primitive types and metadata.

Node builders

The `NodeContainer` contains an `INodeBuilder` member and provides a default implementation for it. So far we didn't had the need to make a custom implementation, since the structure of the graphs themselves is pretty stable.

However, the `INodeBuilder` interface presents an `INodeFactory` member which we override. This factory allows to customize the nodes to be constructed.

The `INodeBuilder` also contains a list of types to be considered as *primitive types*, which means that even if the type contains members or is a reference type, it will be, in term of graph, considered as a primitive value and won't be expanded.

Nodes

There are 3 types of nodes in Quantum:

- `ObjectNode` are node corresponding to an object that is a reference type. They can contain members (properties, fields...), and items (collection).

- `BoxedNode` are a special case of `ObjectNode` that handles `struct`. They are able to write back the value of the struct in other nodes that reference them
- `MemberNode` are node corresponding to the members of an object. If the value of the member is a class or a struct, the member will also contain a reference to the corresponding `ObjectNode`.
- `ObjectNode` that are representing a collection of class/struct items will also have a collection of reference to target nodes via the `ItemReferences` property.

Each node has some methods that allow to manipulate the value it's wrapping. `Retrieve` returns the current value, `Update` changes it. Collections can be manipulated with the `Add` and `Remove` methods (and a single item can be modified also with `Update`).

Events

Each node presents events that can be registered to:

- `PrepareChange` and `FinalizeChange` are raised at the very beginning and the very end of a change of the node value. These events are internal to Quantum.
- `MemberNode`s have the `ValueChanging` and `ValueChanged` events that are raised when the value is being modified.
- `ObjectNode` have `ItemChanging` and `ItemChanged` events that are raised when the wrapped object is a collection, and this collection is modified.

The arguments of these events all inherits from `INodeChangeEventArgs`, which allows to share the handlers between collection changes and member changes.

Finally, Quantum nodes are specialized for assets, where the implementation of the support of override and base is. These specialized classes also present `OverrideChanging` and `OverrideChanged` event to handle changes in the override state.

AssetPropertyGraph

Concept

We use Quantum nodes mainly to represent and save the properties of an asset. The AssetPropertyGraph is a container of all the nodes related to an asset, and describes certain rules such as which node is an object reference, etc.

Asset references

When an asset needs to reference another asset, it should never contains a member that is of the type of the referenced asset. Rather, the type of the member should be the type of the *Content* corresponding to the referenced asset.

Node listener

A node listener is an object that can listen to changes in a graph of node (rather than an individual nodes). The base class is `GraphNodeChangeListener`, and this class must define a visitor that can visit the graph of nodes to register, and stop at the boundaries of that graph.

Object references

In many scenarios of serialization (in YAML, but also in the property grid where objects are represented by a tree rather than a graph), we need a way to represent multiple referencers of the same object such a way that the object is actually expanded at one unique location, and shown/serialized as a reference to all other locations. We introduce the concept of **Object references** to solve this issue.

By design, only objects implementing the `IIdentifiable` interface can be referenced from multiple locations from the same root object. But right now they can only be referenced from the same unique root object (usually an `Asset`). Later on we might support *cross-asset references* but this would require to change how we serialize them.

There are two methods to implement to define if a node must be considered as an object reference or not:

- one for members of an object: `IsMemberTargetObjectReference`
- one for items of a collection: `IsTargetItemObjectReference`

Node presenters

Node presenters are objects used to present the properties of an object to a view system, such as a property grid. They transform a graph of nodes to a tree of nodes, and contains metadata to be consumed by the view. The resulting tree is slightly different from the graph. When an object A contains a member that is an object B that contains a property C, the graph will look like this:

```
ObjectNode A --(members)--> MemberNode B --(target)--> ObjectNode B --(members)--> MemberNode C
```

the corresponding tree of node presenters will be:

```
RootNodePresenter A --> MemberNodePresenter B --> MemberNodePresenter C
```

There is also a `ItemNodePresenter` for collection. On the example above, if B is instead a collection that contains a single item C, the graph would be:

```
ObjectNode A --(members)--> MemberNode B --(target)--> ObjectNode B --(items)--> ObjectNode C
```

the corresponding tree of node presenters will be:

```
RootNodePresenter A --> ItemNodePresenter B --> MemberNodePresenter C
```

Node presenter are constructed by a `INodePresenterFactory` in which `INodePresenterUpdater` can be registered. A `INodePresenterUpdater` allows to attach metadata to nodes, and re-organize the hierarchy in case it want to be presented differently from the actual structures (by inserting nodes to create category, bypassing a class object to inline its members, etc.). `INodePresenterUpdater` have two methods to update node:

- `void UpdateNode(INodePresenter node)` is called on **each** node, after its children have been created. But it's not guaranteed that its siblings, or the siblings of its parents, will be constructed.
- `void FinalizeTree(INodePresenter root)` is called once, at the end of the creation of the tree, and only on the root. Here it's guaranteed that every node is constructed, but you have to visit manually the tree to find the node that you want to customize.

Node presenters listens to changes in the graph node they are wrapping. In case of an update, the children of the modified node are discarded and reconstructed. `UpdateNode` is called again on all new children, and `FinalizeTree` is also called again at the end on the root of the tree. Therefore, you have to be aware that an updater can run multiple time on the same nodes/trees.

Metadata can be attached to node presenters via the `NodePresenterBase.AttachedProperties` property containers. These metadata are exposed to the view models as described in the section below.

Commands can also be attached to node presenters. A command does special actions on a node, in order to update it. Node presenter commands implements the `INodePresenterCommand` interface. A command is divided in three steps, in order to handle multi-selection:

- `PreExecute` and `PostExecute` are run only once, for a selection of similar node presenters, before and after `Execute` respectively.
- `Execute` is run once per selected node presenter.

Node view models

The view models are created on top of node presenters. Each node presenter has a corresponding `NodeViewModel`. In case of multi-selection, a `NodeViewModel` can actually wrap a collection of node presenters, rather than a single one.

Metadata (ie. attached properties) are also exposed from the node presenter to the view via the view model, assuming they are common to all wrapped node presenter, if not, it is possible to add a `PropertyCombinerMetadata` to the property key to define the rule to combine the metadata. The default behavior for combining is to set the value to `DifferentValues` (a special object representing different values) if the values are not equals.

Commands are also exposed. They are added to the view model, combined depending on their `CombineMode` property. They are transformed into WPF commands by being wrapped into a `NodePresenterCommandWrapper`.

All members, attached properties, and commands of node view models are exposed as `dynamic` properties, and can therefore be used in databinding.

All node view models are contained in an instance of `GraphViewModel`. A `GraphViewModelService` is passed in this object that acts as a registry for the node presenter commands and updaters that are available during the construction of the tree.

Template selector

In order to be presented to the property grid, a proper template must be selected for each NodeViewModel. The `TemplateProviderSelector` object picks the proper template by finding the first registered one that accept the given node. Templates are defined in various XAML resource dictionaries, the base one being `DefaultPropertyTemplateProviders.xaml`. There is a priority mechanism that uses an `OverrideRule` enum with four values: `All`, `Most`, `Some`, `None`. One template can also explicitly override the other with the `OverriddenProviderNames` collection. The algorithm that picks the best match is in the `CompareTo` method of `TemplateProviderBase`.

There is actually 3 levels of templates for each property. `PropertyHeader` and `PropertyFooter` represent the section above and the section below the expander that contains the children properties. In the default implementation (`DefaultPropertyHeaderTemplate` and most of its specializations), the header presents the left part of the property (the name, sometimes a checkbox...), and use the third template category, `PropertyEditor`, for the right side of the property grid.

Bases

The base-derived concept and the override are stored in specialized Quantum nodes that implements `IAssetNode`. Properties (as well are items of collections) are automatically overridden when `Update`/`Add`/`Remove` methods are called. Some methods are also provided to manually interact with overrides, but it should not be used directly by users of Quantum.

Node linker

`GraphNodeLinker` is an object that link a given node to another node. It has two main usages: it links objects that are game-side in the scene editor to their counterpart asset-side, and they also link a node to its base if it has one.

The `AssetToBaseNodeLinker` is used to do that. It is invoked at initialization, as well as each time a property changes. It has a `FindTarget` method and `FindTargetReference`, which basically resolve, when visiting the derived graph, which equivalent node of the base graph corresponds to it.

This linker is run from the `AssetPropertyGraph` that can then call `SetBaseNode` to actually link the nodes together.

Reconciliation with base

Each time a change occurs in an asset, all nodes that have the modified nodes as base will call `ReconcileWithBase`. This method visits the graph, starting from the modified properties, and "reconcile" the change. The method is a bit long but well commented. The principle is, for each node, to detect first if something should be reconciled, and if yes, find the proper value (either cloning the value from the base, or find a corresponding existing object in the derived) and set it.

`ReconcileWithBase` is also called at initialization to make sure that any desynchronization that could happen offline is fixed.

Future

Undo/redo

The undo/redo system currently records only the change on the modified object, and rely on `ReconcileWithBase` to undo/redo the changes on the derived object. This is not an ideal design because there are a lot of consideration to take, and a lot of special cases.

What we would like to do is:

- record everything that changes, both in derived and in base nodes
- disbranch totally automatic propagation during an undo/redo

This design was not possible initially, and I'm not sure it is possible to do now - it's possible to hit a blocker when implementing it, or that it requires a lot of refactoring here and there before being doable.

Dynamic nodes

Currently we still expose the real asset object in `AssetViewModel`, which it should never, in the editor, be modified out of Quantum node. Also, manipulating Quantum node is quite difficult sometimes due to indirection with target nodes, and access to members.

```
var partsNode = RootNode[nameof(AssetCompositeHierarchy<TAssetPartDesign, TAssetPart>.Hierarchy
partsNode.Add(newPart);
```

Ideally, we would like to use the `DynamicNode` objects (currently broken) to manipulate quantum nodes:

```
dynamic root = DynamicNode.Get(RootNode);
root.Hierarchy.Parts.Add(newPart)
```

If this is done properly, `AssetViewModel.Asset` could be turned private, and `AssetViewModel` could just expose the root dynamic node, which would allow to seemlessly manipulate the asset through a `dynamic` object.

Contributing to documentation

This documentation serves as a comprehensive guide to help you navigate and contribute to the **Stride Docs** website.

If you're looking to make minor changes, such as adding or updating a manual, tutorial or page, or fixing a typo, feel free to jump straight to the Content Updates section.

For more extensive updates 🤯🤦 or for a deeper understanding of the docs website project, we recommend exploring all the sections provided. Happy browsing and contributing!

Here are the technologies we use to build our website:

- Docfx (static site generator)
  - A specific version of Docfx is utilized in GitHub Actions, one that has been thoroughly tested. Should you wish to upgrade this version, please ensure it is properly tested before implementation.
- Markdown
- Mustache template engine (Docfx dropped Liquid template engine support)
- Bootstrap
- Emojis (because why not? 😎)
- HTML, JavaScript, CSS, JSON
- PowerShell scripts
- GitHub Actions (CI/CD)
  - Our GitHub Actions are already configured for deploying to both staging and release environments.
  - For personal testing or demonstration purposes, you may need to set up your own GitHub Actions. This is especially useful for showcasing proposed changes to maintainers for their approval. For guidance on this, refer to our Deployment to GitHub Pages guide.

Dependencies

Various Stride systems rely on content fetched and processed from either the Stride website or the Stride Docs website. It's crucial to ensure that the following links remain active and accessible. Please refrain from removing or altering these links unless the dependent systems have been updated accordingly to accommodate any changes.

1. https://doc.stride3d.net/latest/en/index.json
   - This JSON file is crucial for integrating the Stride Docs search functionality with the Stride Website. It ensures that search results are comprehensive, including relevant information from both the Stride website and Stride Docs.
2. https://doc.stride3d.net/latest/en/ReleaseNotes/ReleaseNotes.md
   - The **Stride Launcher** utilizes this file when you click a release notes button.

3. https://doc.stride3d.net/latest/en/diagnostics/index.html
    - Diagnostic warnings in the Stride IDE reference pages in the Stride Docs - Diagnostics section. This ensures that users can quickly find detailed explanations and potential solutions for any issues encountered.
4. https://doc.stride3d.net/latest/en/studio_getting_started_links.txt
    - The **Stride Launcher** is using this file in `Urls.Designer.cs`.

Generation Pipeline

Introduction

As of now, **Docfx** does not natively support the generation of multi-language and multi-version documentation. To address this limitation, the Stride team has developed a PowerShell script. Initially, separate scripts were created for each language; however, these have since been consolidated into a single script named `BuildDocs.ps1`. This unified script is capable of generating documentation in all supported languages.

The script serves two main purposes:

- It features a non-interactive mode, utilized by the Continuous Integration/Continuous Deployment (CI/CD) pipeline to automatically generate documentation for all languages and the most recent version, eliminating the need for user intervention.
- It also offers an interactive command-line UI, allowing users to select which languages they wish to generate documentation for.

A Simplified Overview

Here's a straightforward explanation of how the documentation generation process works.

The `/en` folder serves as the repository for the primary documentation files. When documentation for another language (e.g., Japanese) is built, the files from `/en` are copied over to a temporary folder, for example, `/jp-tmp`. This ensures that the non-English versions will contain all the files present in the `/en` folder. Files that have been translated (found in folders like `/jp`) will overwrite their English counterparts in the temp folder `/jp-tmp`.

Docfx is invoked multiple times, once for each language, to create the documentation. The generated documents are stored in the `_site` folder, organized according to the latest version information obtained from `version.json`. For example:

```
/_site/4.1/en
/_site/4.1/jp
```

Docfx Files Processed

This section outlines the file processing carried out by Docfx during the documentation generation:

- **Table of Contents (TOC) Files:** 7 files processed
- **Assets:** 1620 items (images, videos, etc.) included
- **Conceptual Files:** 358 files processed, resulting in 304 HTML files
- **Warnings (No API Metadata):** 44 instances encountered

- **Warnings (API Metadata):** 200 instances of missing or incorrect references
- **API Files:** 2825 files processed, resulting in 2133 HTML files

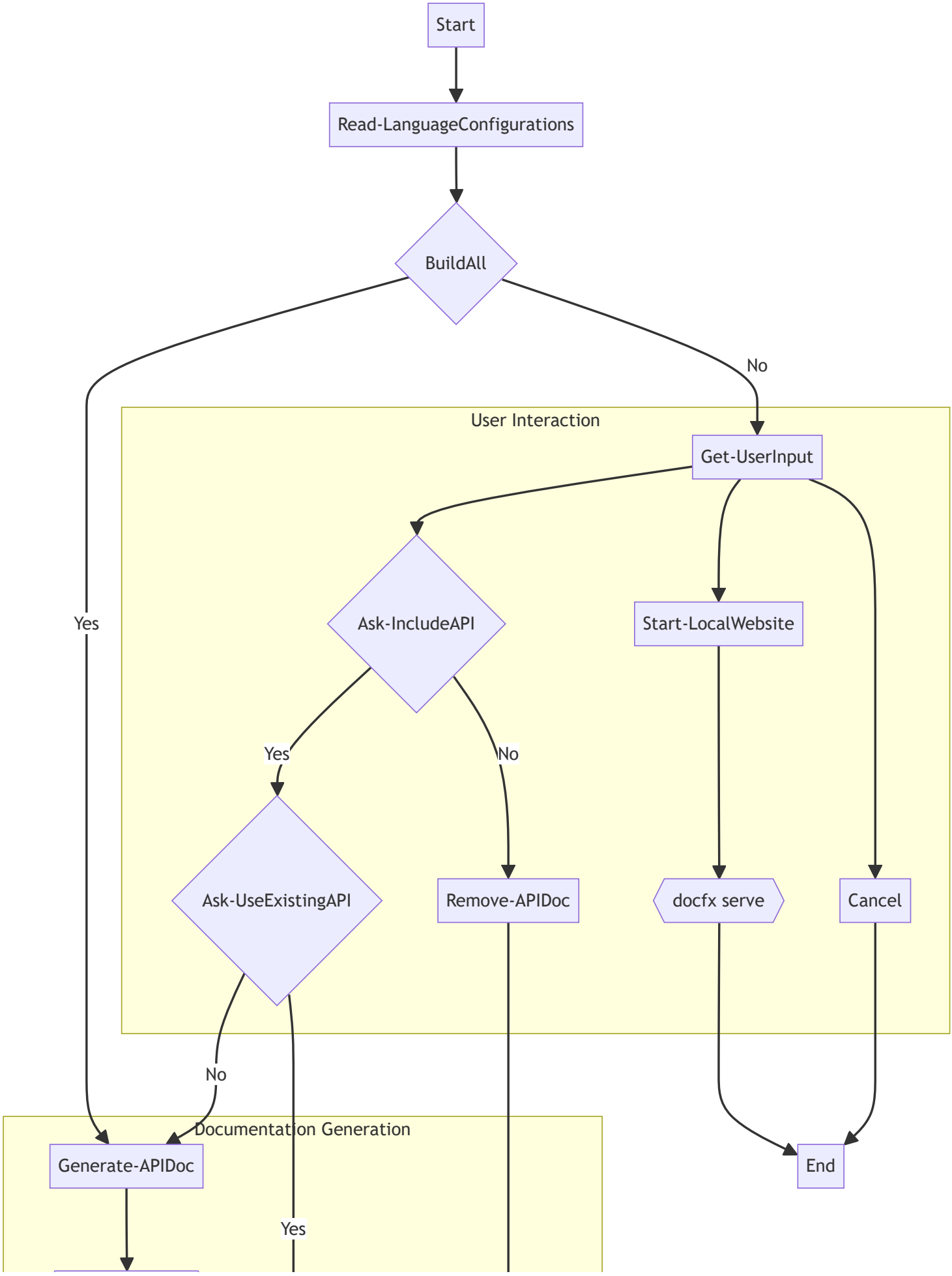
Docs Build Workflow

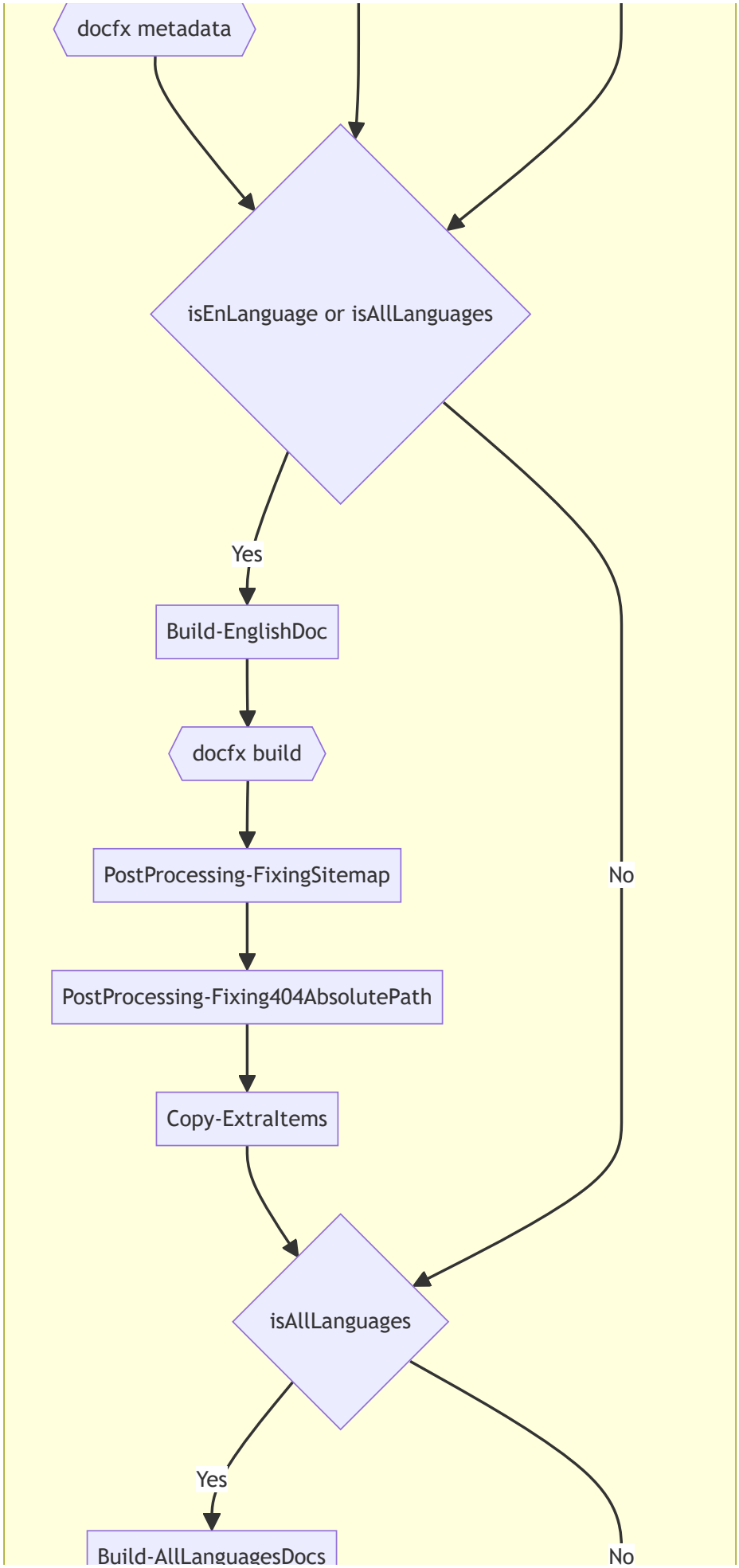
In this part, we elaborate on the individual steps involved in the documentation build workflow for the Stride Docs project.

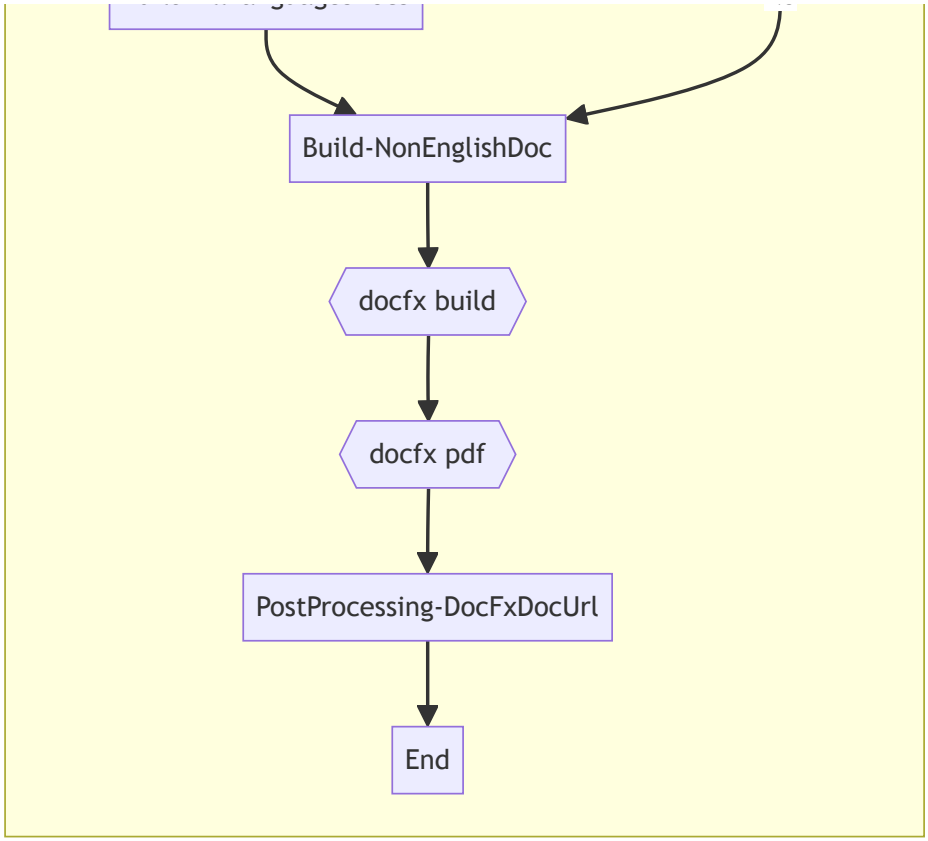
- **Start**
  - Initiates the workflow by reading the `$BuildAll` parameter.
    - If set to 'Yes', it proceeds to generate all languages and the Stride API automatically, which is particularly useful for CI/CD.
    - If set to 'No', it will prompt the user to select languages through an interactive command-line UI.
  - Sets the `$Version` parameter based on the `-Version` command-line argument or fetches it from `version.json` if the argument is not provided.
- **Read-LanguageConfigurations**
  - Reads `languages.json` to identify which languages should be generated.
- **BuildAll**
  - Pre-configures some variables for non-interactive mode, effectively skipping the `Get-UserInput` step.
- **Get-UserInput**
  - In interactive mode, this step prompts the user to choose the languages to generate, as well as whether to launch a local web server.
- **Ask-IncludeAPI**
  - Further queries if the user wants the Stride API included in the documentation build.
- **Ask-UseExistingAPI**
  - Queries if the user wants to re-use already generated Stride API yml files.
- **Start-LocalWebsite**
  - If selected, launches a local web server to host the generated website.
- **Generate-APIDoc**
  - Executes `docfx.exe` to generate the metadata needed for the Stride API documentation.
- **Remove-APIDoc**
  - Removes the generated API metadata.
- **Build-EnglishDoc**
  - Uses `docfx.exe` to build the English documentation, incorporating the Stride API documentation if metadata is available.
- **PostProcessing Steps**
  - PostProcessing-FixingSitemap
    - Adjusts the `sitemap.xml` to use '/latest/en' paths, allowing the most current version to maintain a consistent URL.

    - PostProcessing-Fixing404AbsolutePath
        - Modifies asset (CSS, JS,) paths in `404.html` to be absolute, as required by IIS for 404 page.
    - Copy-ExtraItems
        - Copies additional items like `versions.json`, `web.config`, `ReleaseNotes.md` and `robots.txt`, while also updating the `%deployment_version%` parameter in the `web.config` file.
- **Build-AllLanguagesDocs**
    - Iterates over all selected languages and triggers the `Build-NonEnglishDoc` function for each.
- **Build-NonEnglishDoc**
    - Executes `docfx.exe` to compile non-English documentation, incorporating Stride API documentation if metadata is present.
- **PostProcessing-DocFxDocUrl**
    - Adjusts HTML tags and GitHub links, removing any `_tmp` suffixes. Also updates GitHub links to English if the translation is unavailable.

Workflow Diagram


Start
Read-LanguageConfigurations
BuildAll
Yes
No
User Interaction
Get-UserInput
Ask-IncludeAPI
Start-LocalWebsite
Yes
No
Ask-UseExistingAPI
Remove-APIDoc
docfx serve
Cancel
No
Documentation Generation
Generate-APIDoc
Yes
End


docfx metadata
isEnLanguage or isAllLanguages
Yes
No
Build-EnglishDoc
docfx build
PostProcessing-FixingSitemap
PostProcessing-Fixing404AbsolutePath
Copy-ExtraItems
isAllLanguages
Yes
No
Build-AllLanguagesDocs


Build-NonEnglishDoc
docfx build
docfx pdf
PostProcessing-DocFxDocUrl
End

Local installation

This guide will walk you through the steps to install the Stride Docs website on your local machine for development purposes. Although we use the Windows operating system for development, the steps should be similar for other operating systems.

Minor updates can be made directly on GitHub. However, for more significant updates that affect multiple pages, we recommend using a local development environment so you can see the impact of your changes beforehand. This is because we use the Docfx static site generator, and in some cases, all pages need to be regenerated. This approach helps you assess your changes before submitting a pull request.

This guide assumes you have a basic understanding of the technologies used in the Stride docs website.

Prerequisites

Before updating the Stride Docs, ensure you are familiar with the following prerequisites:

1. Familiarity with the command line
2. **.NET SDK 8.0 or higher:** You can download the installer from the .NET SDK website
   - If .NET SDK is already installed, ensure you have version 8.0 or higher. You can check your version by running `dotnet --info` in a terminal.
3. **Git installed:** You will need Git for version control. If you don't have Git installed, you can download it from the Git website
4. **Development IDE of choice:** Choose an Integrated Development Environment (IDE) that you're comfortable with for development. Although there are various popular choices, such as Visual Studio, Visual Studio Code, and others, this guide will focus on using **Visual Studio**, as it is the primary IDE for the Stride project, and as of writing, we use **Visual Studio 2022**. You can download the free Community edition from the Visual Studio website

Installation Steps

1. ❓You might want to create an issue so we can track your contribution and avoid duplicate work. If you're unsure whether your contribution is needed, feel free to create an issue and ask
2. 🍴 Fork the repository by navigating to the Stride Docs repository and clicking the **Fork** button in the top-right corner
3. 📥 Clone your forked repository using the following command, replacing `your-username` with your GitHub username: `git clone https://github.com/your-username/stride-docs.git`
   - 💡**Tip:** It's a good idea to create a new branch for each feature or bug fix you work on. This helps keep your forked repository organized and makes it easier to manage multiple pull requests
4. Make sure you have also Stride repo cloned on **the same level** as stride-docs, read more about it here

    - This repo is needed for API documentation generation
5. 📁 Go to the project folder `cd stride-docs`
6. 🚀 Let's start with the **Docfx**

Enter the following command to install the latest docfx

```
dotnet tool install -g docfx
```

Or check the installed version is at least `2.74.1`

```
docfx --version
```

Other options

Update to the latest Docfx

```
dotnet tool update -g docfx
```

Install a specific version of Docfx

```
dotnet tool update -g docfx --version 2.74.1
```

Uninstall Docfx if you need to downgrade

```
dotnet tool uninstall -g docfx
```

Running the Development Server

We've created a PowerShell script BuildDocs.ps1 with a context menu where you can select the language, include the API build, and run the development server.

1. 🚀 Run `run.bat` in the command line to start the script
2. 📋 You will see the following self-explanatory menu:

```
Please select an option:

  [en] Build English documentation
  [jp] Build Japanese documentation
  [all] Build documentation in all available languages
  [r] Run local website
  [c] Cancel
```

```
Your choice:
```

3. 🌐 Choose to build the documentation in the language of your preference
   - Select `[n]` for no API build
4. 🖥️ If you select `[r]`, the documentation site will open automatically in your browser `http://localhost:8080/en/index.html`
   - If you built the documentation in a language other than English, you'll need to manually change the language in the URL
5. 💻 Open the project in Visual Studio by opening the `Stride.Docs.sln` solution file, or use the IDE of your choice
6. 🔄 After saving the updated file, you will need to rebuild the documentation by running the script again
7. 😃 Happy coding!

Let's update the content now!

Documentation content

Content Updates

If you want to contribute and update the website, please follow the instructions below.

Small updates can be done directly in the GitHub web interface, for bigger updates the local development environment is required, which is described in the Installation section.

You can use any text editor to make changes. If you are using **Visual Studio**, you can open `Stride.Docs.sln` solution file in the root of the repository and start making your updates directly from this IDE.

You are always welcome to create an issue to discuss your changes before you start working on them.

Small Updates

Creating an issue is not required for small updates, but it is recommended to let others know what you are working on. If you are not sure whether your update is small or not, please create an issue first.

What is a small update?

We can define small updates as changes to the content of the website:

- Update the content of an existing page (manual, tutorial or release note, ..)
- Add a new manual or tutorial or any new content
- Fix a typo

Steps

> **NOTE**
> This guide assumes that you are already familiar with updating files on GitHub.

For the following instructions, use the Stride Docs GitHub repository:

1. Go to the repository
2. Locate the file you wish to edit
3. Click the `Edit this file` (pencil) icon in the top right corner
4. If prompted, fork the repository by clicking `Fork this repository`
5. Make your changes to the file, then write a brief commit message describing the changes
6. Click on the `Propose changes` button
7. On the next screen, click the `Create pull request` button
8. Provide a title and description for your pull request, and click on `Create pull request` again

9. Wait for the review and merge

Major Updates

Creating an issue is **required** for major updates, so that others can comment on your changes and provide feedback.

Major updates can be defined as significant changes to the website's design, where it's beneficial to preview the impact of your changes to ensure they achieve the desired result. This may include:

- Update Docfx version
- Modifying layouts
- Revamping design elements

Start by setting up your local development environment, as described in the Installation section. After making and testing your changes locally, you should create a pull request to merge your changes into the `master` branch.

When submitting a pull request, especially for substantial changes, it's recommended to include **screenshots** or a link to your local deployment. This approach helps maintainers visualize and assess your proposed changes more effectively. If you prefer to use GitHub infrastructure for your demonstrations, refer to our Deployment to GitHub Pages guide for instructions on deploying via GitHub Actions.

Manual

These pages contain information about how to use Stride, an open-source C# game engine.

> **IMPORTANT**
>
> **SEO Note:** Ensure that the file name includes essential keywords related to the content of the article. This is crucial because the file name dictates the URL of the content page, which plays a significant role in search engine optimization (SEO).

Creating New Manual Page

1. Create a new file in the `manual` folder, in the already existing folders (e.g. animation, audio, ..) or create a new folder in the `manual` folder.
    - If you created a new folder, make sue that you create also `index.md` file in this folder.
2. Use any existing page as a template for the new page.
3. Update `toc.yml` (or `toc.md`) file in the `manual` folder to include the new page or folder. The `toc.yml` file contains the table of contents for the manual pages, which is displayed on the left side of the manual pages. These pages are also included in the optionally generated PDF file.

Naming Convention

Observe existing pages and folders for the naming convention.

Media

You can observe that existing folders might have a `media` folder. This folder contains images and videos used in the manual pages. You can use this folder or create a new one in your folder. If possible make sure that images are `.webp` format and videos are `.mp4` format.

Tutorial

These pages contain tutorials on how to use Stride, an open-source C# game engine.

Creating New Tutorial Page

1. Create a new tutorial folder in the `tutorial` folder.
2. Create a new `index.md` file in this folder. Observe existing tutorials for the content of this file.
3. Create markdown files for each step of the tutorial. Observe existing tutorials structure for the content of these files.
4. Update `toc.yml` file in the `tutorial` folder to include the new tutorial folder. The `toc.yml` file contains the table of contents for the tutorial pages, which is displayed on the left side of the tutorial pages.

Naming Convention

Observe existing pages and folders for the naming convention.

Media

You can observe that existing tutorials have a `media` folder. This folder contains images. If possible make sure that images are `.webp` format. The videos should be uploaded to YouTube and embedded in the tutorial pages.

Other Sections

In addition to the Manual and Tutorial sections mentioned above, the same principles apply to both existing and new sections. Follow the established formats and conventions to ensure consistency and clarity throughout the documentation.

Shortcodes and Includes

Docfx supports additional markdown syntax to enrich content. These syntaxes are specific to Docfx and **may not render** correctly on other platforms, like GitHub.

For more information, read the Docfx documentation on markdown, shortcodes and includes. Some commonly used features include:

- **Alert**: These are block quotes that render with distinct colors and icons, highlighting the importance or nature of the content
- **Video**: Embed video content directly into your documentation
- **Image**: Insert images to enhance the visual aspect of the documentation
- **Math Expressions**: Integrate mathematical notations and expressions
- **Mermaid Diagrams**: Embed mermaid diagrams for flowcharts and other graphical representations
- **Include Markdown Files**: Include content from other markdown files seamlessly
- **Code Snippet**: Insert code snippets for better clarity and demonstration
- **Tabs**: Organize content into tabbed sections for improved readability

Web Assets

Our main web assets include:

- `template/partials/affix.tmpl.partial` - Currently not functioning
- `template/partials/footer.tmpl.partial` - Currently not functioning
- `template/public/main.css` - Contains minor Bootstrap CSS overrides
- `template/public/main.js`:
  - Sets the top navigation icons, such as GitHub, Discord, Twitter
  - Injects the Stride Docs version selection above the filter in the side navigation
  - Injects the Stride Docs language selection into the top navigation
- `docfx.json` - The HTML footer is included in the `_appFooter` section

Styling

Bootstrap Customization

We utilize the `modern` template provided by Docfx, which employs the Bootstrap framework, version **5.3**. This includes the dark theme, enabled by Docfx.

> **IMPORTANT**
>
> Prioritize the use of Bootstrap's inherent styling before integrating any custom styles. You should be familiar with Bootstrap Utilities which help you to achieve most of the styling requirements.

CSS Guidelines

Our goal is to write minimal CSS code to keep the website lightweight, leveraging the Bootstrap framework to the fullest extent possible.

Submitting your Changes

Assuming you have made all necessary changes and tested them on the development server, you can submit a pull request to the `master` branch. The pull request will be reviewed and merged by the website maintainers.

Steps to contribute your updates:

1. Commit your changes to your forked repository:
    - Commit the changes with a meaningful message
    - Push the changes to your forked repository
2. Create a pull request to the main repository:
    - You can create a pull request from your forked repository by navigating to Pull requests page and click **New pull request** button
    - Select the **master** branch as the base branch and your branch as the compare branch
    - Click **Create pull request** button

Once your pull request has been reviewed and approved, your changes will be merged into the main repository and deployed to the website.

Documentation Roadmap

This document outlines a proposed roadmap and an ongoing development plan for our Stride Docs website.

- **Address Existing Issues**: Prioritize resolving issues listed in the Issues section on GitHub.
- **Image Optimization**: Convert existing images to the WebP format to enhance website performance.
- **Content Enhancement**: Implement improvements across all sections of the documentation to ensure clarity, accuracy, and comprehensiveness.
- **Guidance for Contributors**: Provide clear instructions for contributors on writing XML comments in C#, which play a crucial role in enhancing the API documentation.

Docfx

Docfx is a static site generator that uses C# as its templating language. It is an exceptionally powerful tool, offering immense flexibility and customization options for creating a documentation website. Moreover, Docfx is user-friendly and easy to learn. This section covers the basics of Docfx configuration for the Stride Docs website, while the creation and updating of content are detailed in our Content section.

After reviewing various static site generator options, we decided to continue using Docfx, particularly in light of the release of the new `modern` Docfx template. This template leverages Bootstrap 5.3 and has recently introduced a dark theme feature.

Packages and Dependencies

Currently, we are not utilizing any additional packages.

Configuration

The configuration for Docfx is located in the `en\docfx.json` file. This file contains all the necessary settings for the Docfx build process.

Contents of the Configuration File:

- **API Sources**: Specifies the Stride path and selected projects for API documentation generation
- **Global Metadata**: Contains global configuration settings for the documentation build
- **File Metadata**: Defines folder sections to be processed for documentation generation, such as Manuals, Tutorials, etc.
- **Resource - Pass Through Files**: Lists files that are copied directly to the output folder without processing
- **Other Configuration**: Explore the file for additional configuration options

For more details on configuration options, visit the Docfx Configuration Documentation.

Global Data

Docfx currently does not support global data like 11ty. At present, *Mustache* can only be used in templates.

Folder Structure

The folder structure plays a vital role in the documentation generation process, as it determines the output of the build. The structure is organized as follows:

Folders

- `.github`: Contains GitHub Action workflows
- `_site`: The output build folder (excluded in `.gitignore` and used for deployment)
- `en`: Contains the English language documentation
- `en\api`: Automatically generated folder from the Stride API
- `en\contributors`: Documentation for contributors
- `en\diagnostics`: Diagnostic pages referenced by Stride solution warnings in the IDE
- `en\examples`: Additional content for C# XML comments, which are merged into API documentation and linked by **uids**
- `en\includes`: Markdown files whose content can be included in multiple `.md` files across the documentation.
- `en\manual`: Documentation for the manual
- `en\media`: Main media assets
- `en\ReleaseNotes`: Documentation for release notes
- `en\template`: Docfx assets for minor template customization, including CSS and JS files
- `en\tutorials`: Documentation for tutorials
- `jp`: Japanese language documentation, translated from the English version (currently not updated)
- `wiki`: GitHub wiki content - Excluded from the build process and used only for wiki deployment. This section will be decommissioned as the content has been moved to Stride Docs.

Files

- `en\*.md`: Markdown content pages
- `en\*.yml`: Table of content files
- `en\.nojekyll`: A flag file for GitHub Actions
- `en\docfx.json`: Docfx configuration file
- `en\filterConfig.yml`: Rules for API exclusion
- `en\languages.json`: Configuration file for languages

Non Docfx Files

- `appsettings.json`: Configuration file for ASP.NET Core.
- `appsettings.Development.json`: Development-specific configuration file for ASP.NET Core.
- `build-all.bat`: Batch file used in GitHub Actions CI/CD to build all documentation using `BuildDocs.ps1`.
- `BuildDocs.ps1`: PowerShell script responsible for building documentation. Refer to pipeline for details.
- `OldDocsFix.ps1`: Temporary PowerShell script for fixing old documentation.
- `Program.cs`: Startup file for ASP.NET Core.
- `run.bat`: Batch file to run `BuildDocs.ps1` in interactive mode.
- `run-fix.bat`: Temporary batch file to run `OldDocsFix.ps1`.
- `Stride.Docs.csproj`: ASP.NET Core project file.
- `Stride.Docs.sln`: ASP.NET Core solution file.

- `Stride.Docs.csproj.user`: User-specific ASP.NET Core project file.
- `versions.json`: Configuration file managing versions of Stride documentation.
- `web.config`: Configuration file for IIS deployment.

> **NOTE**
>
> This project includes the Visual Studio solution `Stride.Docs.sln`, allowing you to edit the files using the Visual Studio IDE.

Layouts

We utilize the default layout provided by the `modern` template, as specified in `docfx.json`.

Includes

All includes are located in the `/_includes` folder. These are reusable markdown snippets that can be incorporated into multiple pages.

Deployment

Our team has explored various deployment options, ultimately selecting the method detailed in this guide for its efficacy. Additionally, for demonstration purposes, you can refer to the Deployment to GitHub Pages section for alternative deployment strategies you can use to showcase your updates.

Deploying to Azure Web Apps (Windows) with IIS

This guide is crafted for individuals who already have access to the Azure subscription. It provides step-by-step instructions for setting up a new Azure Web App, specifically tailored for staging environments. Note that the process for setting up a production environment is similar, but requires a distinct web app name.

Deployments to Azure Web Apps are automated through GitHub Actions, forming an integral part of our Continuous Integration/Continuous Deployment (CI/CD) process. The CI/CD pipeline is configured to automatically trigger deployments upon merging changes into either the `staging` or `release` branches.

> **NOTE**
> The deployment process outlined here is already established and running, hosted on Azure and sponsored by the .NET Foundation. This guide serves primarily as a reference for maintainers in the event that a new deployment setup is required.

Setting up a new Azure Web App

Follow these instructions carefully to establish your Azure Web App in a staging environment. For deploying in a production environment, replicate these steps with an alternate web app name for differentiation.

1. Navigate to the Azure Portal
2. Select **Create a resource**
3. Choose **Create a Web App**
4. In the Basic Tab
    - Choose your existing subscription and resource group
    - Under Instance Details, enter:
        - Name: **stride-docs-staging**
        - Publish: **Code**
        - Runtime stack: **ASP.NET V4.8**
        - OS: **Windows**
        - Region: as the current web

        - Pricing Plan - An existing App Service Plan should appear if the region and resource group match that of the existing web app. Currently we use **Standard S1**.
        - Click **Next**
5. In the Deployment Tab - This step can be completed later if preferred.
    - Enable Continuous deployment
    - Select account, organisation `Stride`, repository `stride-docs` and branch `staging`
    - Click **Next**
6. In the Monitoring Tab
    - Leave all settings as default
    - Click **Next**
7. Monitoring Tab
    - Disable Application Insights - This is not needed at this stage
    - Click **Next**
8. In the Tags Tab
    - Leave this blank unless you wish to add tags
    - Click **Next**
9. In the Review Tab
    - Review your settings
    - Click **Create**
    - The GitHub Action will be added to the repository and run automatically. It will fail at this stage, but this will be resolved in the subsequent steps.

> **CAUTION**
>
> If you have completed the **Deployment Tab** process, ensure that the deployment profile includes the **DeleteExistingFiles** property. This property may need to be set to `False` or `True` depending on the specific requirements of your deployment. For instance, Stride Docs deployment retains files from previous deployments, allowing multiple versions like `4.2`, `4.1`, etc., to be maintained. Adjust this setting based on your deployment needs.

Adjusting the Web App Configuration

1. Proceed to the newly created Web App
2. Click on **Configuration**
3. Select **General Settings**
4. Change the `Http version` to **2.0**
5. Change `Ftp state` to **FTPS only**
6. Change `HTTPS Only` to **On**
7. Click **Save** to apply the changes

Modifying the GitHub Action

The previous step will have added a GitHub Action to your repository, which might fail initially. To address this, you need to modify the GitHub Action:

1. Navigate to the repository
2. Select **Actions**
3. You have the option to stop the currently running action
4. Locate the new GitHub Action file *(stride-docs/blob/master/.github/workflows/some-file-name.yml)* that was automatically generated by Azure Portal. We need to extract the `app-name` and `publish-profile` values from it and disable the push trigger.
   - To disable the push trigger, retain only **workflow_dispatch** (manual trigger) as shown below:

```
on:
#  push:
#    branches:
#      - staging
    workflow_dispatch:
```

5. Open the `stride-docs-staging-azure.yml` workflow and update it with the values obtained in the previous step. Save your changes.
6. This workflow might also need to be added to the `master` branch if it is not already present.
7. Execute the workflow `stride-docs-staging-azure.yml`. Ensure you select the correct branch `staging` and click **Run workflow**. This action will deploy the website to the Azure Web App.

GitHub Actions

- `stride-website-github.yml`: Enables manual deployment to GitHub Pages in a forked repository, primarily for showcasing updates.
- `stride-docs-release-azure.yml`: Automates deployment to production upon merging changes into the `release` branch, with a manual trigger option also available.
- `stride-docs-release-fast-track-azure.yml`: Provides manual deployment to production, bypassing the creation of artifacts.
- `stride-docs-staging-azure.yml`: Facilitates automatic deployment to staging when changes are merged into the `staging` branch, and includes a manual trigger option.
- `stride-docs-staging-fast-track-azure.yml`: Allows for manual deployment to staging, skipping the creation of artifacts.
- `stride-website-wiki.yml`: Deploys automatically to the GitHub Wiki upon changes being pushed to the `wiki` folder, with a manual trigger feature also available.

Deployment to GitHub Pages

To showcase your updates, especially helpful for design changes pending review, you can deploy the docs website either to your infrastructure or to GitHub Pages, a free hosting service. Once deployed, share the link with us for review.

Prerequisites

In your `stride-docs` repository:

1. Navigate to **Settings** → **Actions** → **General** → **Workflow Permissions**
    - Choose **Read and write permissions**

Run GitHub Action

1. Go to **Actions**, select **Build Stride Web for GitHub Staging**
    - Click **Run workflow**; you may optionally select a branch
2. Monitor the build logs while the action is in progress
3. Upon successful build, a `gh-pages` branch will be created
4. Navigate to **Settings** → **Pages**
    - Choose the `gh-pages` branch with the root option and click **Save**
5. After saving, an internal GitHub Action **pages build and deployment** is automatically created and triggered, deploying the content to the GitHub Pages website
6. The website will be accessible at `https://[your-username].github.io/stride-docs`

Major Release Workflow

Assuming the transition is from version `4.1` to `4.2`, and that the Stride source code has been updated to the corresponding .NET version, follow these steps. Note that some steps can be executed at a later stage if needed.

1. Duplicate `ReleaseNotes\ReleaseNotes.md` and rename the copy to `ReleaseNotes-4.1.md`
2. Update `ReleaseNotes.md`:
   - Change the content title to `4.2`
   - Replace the content with the new release notes for version `4.2`
   - GitHub Release can be used to generate a list **What's Changed**, once the new tag was added
3. Modify `ReleaseNotes\toc.yml`
   - `name: 4.2 release notes` with `href: ReleaseNotes.md`
   - `name: 4.1 release notes` with `href: ReleaseNotes-4.1.md`
4. In `en\docfx.json`
   - `_appFooter`: Increase the version number
   - Change `TargetFramework` to the current framework version being used. Ensure to test this step locally
5. Edit `versions.json`
   - Under `versions`, add the new version `4.2`
6. For GitHub Actions deployment update `*.yml` files in the `.github\workflows\` folder
   - `dotnet-version:` Update to the related .NET version

The `BuildDocs.ps1` script will manage the deployment to the `4.2` folder while maintaining accessibility to previous versions. Note, that the deployment profile must be set to not delete existing items.

Other locations to update

1. Update README.md in the Stride repo, Building from source - Prerequisites section, bump .NET version
2. Modify `contributors\documentation\installation.md`
   - Update SDK version

Troubleshooting and FAQ

Known Issues

1. **Sponsor Page - Widget Incompatibility with dark theme:** Widgets on the Sponsor Page currently do not support the dark theme. As a workaround, we can either fetch data from https://opencollective.com/stride3d/members/all.json and render it before deployment or make it dynamic. Both options would give us more control over the content and design, and allow for better compatibility with the dark theme in the future.
2. **Search Page - Lack of pagination:** At present, the Search Page does not have pagination, which limits the maximum number of search results displayed to 100. To resolve this issue, we can implement a pager in JavaScript. This would enable users to navigate through multiple pages of search results, providing a more comprehensive view of the available content.

Common Issues and Solutions

Any issue should be added to Stride Website GitHub issues so it can be tracked and elaborated by the community.

Frequently Asked Questions

Q: I just want to fix a typo in a post. Do I need to follow your installation steps?

A: *No, you can fix the typo directly on the GitHub website. However, you will still need to fork the repo, make your update on the main branch or a new branch, and then create a pull request. You can follow this guide for minor updates.*

Contributing to the Stride website

This documentation serves as a comprehensive guide to help you navigate and contribute to the **Stride website**.

If you're looking to make minor changes, such as adding or updating a post or page, or fixing a typo, you can jump straight to the Content Updates section.

For more extensive updates 🤯🤦 and a deeper understanding of the website project, we recommend exploring all the sections provided. Happy browsing and contributing!

Technologies we use to build our website:

- Eleventy (static site generator)
- Markdown
- Mainly Liquid and a bit Nunjucks (template engines)
- Bootstrap
- Font Awesome
- HTML, JavaScript, CSS, SCSS, and JSON
- GitHub Actions (CI/CD)
    - Our GitHub Actions are already configured for deploying to both staging and release environments.
    - For personal testing or demonstration purposes, you may need to set up your own GitHub Actions. This is especially useful for showcasing proposed changes to maintainers for their approval. For guidance on this, refer to our Deployment to GitHub Pages guide.

Dependencies

Various Stride systems rely on content fetched and processed from either the Stride website or the Stride Docs website. It's crucial to ensure that the following links remain active and accessible. Please refrain from removing or altering these links unless the dependent systems have been updated accordingly to accommodate any changes.

1. https://www.stride3d.net/legal/privacy-policy/
    - This link is integral to the **Stride Installer**. It provides users with transparent information about data handling and privacy considerations associated with using Stride.
2. https://www.stride3d.net/feed.xml
    - The **Stride Launcher** utilizes this feed to keep users updated with the latest news, updates, and announcements from the Stride community.
3. https://doc.stride3d.net/latest/en/index.json
    - This JSON file is crucial for integrating the Stride Website's search functionality with the Stride Documentation. It ensures that search results are comprehensive, including relevant information from both the Stride website and Stride Docs.

Local installation

This guide will walk you through the steps to install the Stride website on your local machine for development purposes. Although we use the Windows operating system for development, the steps should be similar for other operating systems.

Minor updates can be made directly on GitHub. However, for more significant updates that affect multiple pages, we recommend using a local development environment so you can see the impact of your changes beforehand. This is because we use the Eleventy static site generator, and in some cases, all pages need to be regenerated. This approach helps you assess your changes before submitting a pull request.

This guide assumes you have a basic understanding of the technologies used in the Stride website.

Prerequisites

Before updating the Stride website, ensure you are familiar with the following prerequisites:

1. Familiarity with the command line
2. **.NET SDK 8.0 or higher:** You can download the installer from the .NET SDK website
   - If .NET SDK is already installed, ensure you have version 8.0 or higher. You can check your version by running `dotnet --info` in a terminal.
3. **Git installed:** You will need Git for version control. If you don't have Git installed, you can download it from the Git website
4. **Development IDE of choice:** Choose an Integrated Development Environment (IDE) that you're comfortable with for development. Although there are various popular choices, such as Visual Studio, Visual Studio Code, and others, this guide will focus on using **Visual Studio**, as it is the primary IDE for the Stride project, and as of writing, we use **Visual Studio 2022**. You can download the free Community edition from the Visual Studio website

Installation Steps

1. ❓You might want to create an issue so we can track your contribution and avoid duplicate work. If you're unsure whether your contribution is needed, feel free to create an issue and ask
2. 🍴 Fork the repository by navigating to the Stride website repository and clicking the **Fork** button in the top-right corner
3. 📥 Clone your forked repository using the following command, replacing `your-username` with your GitHub username: `git clone https://github.com/your-username/stride-website.git`
   - 💡**Tip:** It's a good idea to create a new branch for each feature or bug fix you work on. This helps keep your forked repository organized and makes it easier to manage multiple pull requests
4. 📂 Go to the project folder `cd stride-website`
5. 🚀 Run `npm install` to install all dependencies

Running the Development Server

1. 🚀 Run `npm start` (`npx @11ty/eleventy --serve`) in the command line to start the development server
2. 📋 You should see many logs in the command line, indicating the progress and displaying any errors
    - ⚠️ A Windows Security warning may appear on the first run (Allow Node.js JavaScript Runtime to communicate on these networks). Click **Allow access**
3. 🌐 Open the site in your browser by navigating to `http://localhost:8080/`
4. 💻 Open the project in Visual Studio by opening the `Stride.Web.sln` solution file, or use the IDE of your choice
5. 🔄 Once you save the updated file, the website will automatically refresh in the browser
6. 😃 Happy coding!

ToDo: Attach a screenshot of the command line output

Let's update the content now!

ASP.NET Core

This static website can also be hosted using ASP.NET Core.

Although we're not currently using the ASP.NET Core website, it remains available for future use. If necessary, we can integrate dynamic ASP.NET Core pages with the static pages generated by Eleventy.

To edit the website through Visual Studio, open the `Stride.Web.sln` solution, which will load the website in the IDE. You can then modify the pages and content and run the website directly from Visual Studio.

During the Visual Studio build process, `npm run build` is executed, generating the static website in the same `_site` folder as previously described. The ASP.NET Core website uses this folder instead of the default `wwwroot`. This customization is specified in the `Program.cs` file.

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    WebRootPath = "_site" // Set the folder where the static files are located (e.g., Eleventy
});
```

Website Content

Content Updates

If you want to contribute and update the website, please follow the instructions below.

Small updates can be done directly in the GitHub web interface, for bigger updates the local development environment is required, which is described in the Installation section.

You can use any text editor to make changes. If you are using **Visual Studio**, you can open `Stride.Web.sln` solution file in the root of the repository and start making your updates directly from this IDE.

You are always welcome to create an issue to discuss your changes before you start working on them.

Small Updates

Creating an issue is not required for small updates, but it is recommended to let others know what you are working on. If you are not sure whether your update is small or not, please create an issue first.

What is a small update?

We can define small updates as changes to the content of the website:

- Update the content of an existing page
- Update the content of an existing blog post
- Add a new page or blog post
- Fix a typo
- Minor navigation or footer update
    - This will update all pages containing the navigation or footer

Steps

> **NOTE**
> This guide assumes that you are already familiar with updating files on GitHub.

For the following instructions, use the Stride Website GitHub repository:

1. Go to the repository
2. Locate the file you wish to edit
3. Click the `Edit this file` (pencil) icon in the top right corner
4. If prompted, fork the repository by clicking `Fork this repository`
5. Make your changes to the file, then write a brief commit message describing the changes

6. Click on the `Propose changes` button
7. On the next screen, click the `Create pull request` button
8. Provide a title and description for your pull request, and click on `Create pull request` again
9. Wait for the review and merge

Major Updates

Creating an issue is **required** for major updates, so that others can comment on your changes and provide feedback.

Major updates can be defined as significant changes to the website's design, where it's beneficial to preview the impact of your changes to ensure they achieve the desired result. This may include:

- Adding new Eleventy shortcodes and Liquid includes
- Updating the Bootstrap library or other libraries
- Modifying layouts
- Revamping design elements

Start by setting up your local development environment, as described in the Installation section. After making and testing your changes locally, you should create a pull request to merge your changes into the `master` branch.

When submitting a pull request, especially for substantial changes, it's recommended to include **screenshots** or a link to your local deployment. This approach helps maintainers visualize and assess your proposed changes more effectively. If you prefer to use GitHub infrastructure for your demonstrations, refer to our Deployment to GitHub Pages guide for instructions on deploying via GitHub Actions.

Creating New Post

To create a new blog post, you can follow one of these methods:

1. Copy an existing post and update the front matter and content. This is the fastest way to get started with a new post
2. Alternatively, create a new file in the `posts` folder, ensuring that the file name follows the appropriate naming convention

Either method will allow you to create a new blog post, so choose the one that best suits your needs.

Post Naming Convention

`YYYY-MM-DD-post-title.md`

Replace `YYYY-MM-DD` with the date of the post and `post-title` with the title of the post.

> **IMPORTANT**
>
> **SEO Note:** Ensure the file title includes essential keywords related to your post's content. This is crucial as the file title dictates the URL of the post, which plays a significant role in search engine optimization (SEO).

Post Front Matter

The file should start with the following front matter:

```
---
title: 'Post title'
# author's id, defined in the _data/site.json
author: vaclav
# optional, if not set, the default tags will be used, tags are merged with the default tags
# you can find all tags in the live site in the /tags/ page
tags: ['Announcement']
# optional, if not set, the default image will be used
# use webp format for best performance, images should be located in the /images/blog/YYYY-MM-DD
image: /images/blog/2023-04/new-home-page.webp
# optional, if true, the post will be featured in the popular section
pupular: true
# permlink is automatically generated based on the file name, but you can override it here
permalink: /blog/2023-04/my-custom-link/ # this is a custom link
---
```

The same example, without the comments:

```
---
title: 'Post title'
author: vaclav
tags: ['Announcement']
image: /images/blog/2023-04/new-home-page.webp
pupular: true
permalink: /blog/2023-04/my-custom-link/
---
```

Default front matter, which is used for all posts, can be found in the `posts/posts.json` file.

```
{
  "layout": "post",
```

```
  "eleventyComputed": {
    "year": "{{ page.date | date: '%Y' }}",
    "modified": "Last Modified"
  },
  "permalink": "/blog/{{ page.fileSlug }}/",
  "tags": [ "blog", "search" ]
}
```

Image

The image specified in the front matter serves dual purposes: It appears in the blog listing at Stride Blog and is used as the **og:image** meta tag for social sharing. Here are three ways to specify this image:

- Not including an image in the front matter will use the default image
- Including an image in the front matter will override the default image. The size of the image should be minimum **1200 x 600px** e.g. `image: /images/blog/2023-04/new-home-page.webp`
- External image URL e.g. `image: https://i.imgur.com/7GVEiSR.jpg`
- If you are looking for Stride specific logo's or icons, have a look at the Figma options

Post Content

Check the previous posts for an example of the post content. Ideally you should use the same format as the previous posts to preserve the consistency of the blog.

You can use shortcodes and includes which are described in the Shortcodes and Includes section.

You can also use majority of the Bootstrap classes in your content if you combine HTML and Markdown.

> **TIP**
>
> We have a folder called `_drafts` where you can store your drafts. These files are not published. Once you are ready to publish your post, you can move it to the `posts` folder.

Excerpt

The excerpt is the first paragraph of the post. Separated from the rest of the content by three dashes `---`. The excerpt is used in the blog post list, meta description and in the RSS feed.

Example

```
---
title: 'Stride 4.1 is Now Live'
author: aggror
```

```
tags: ['Tutorials','Release', 'Graphics']
---

Stride contributors are proud to announce a new release now running on .NET 6 supporting the la

---

Additional content goes here...
```

Creating New Page

To create a new page, create a new file in the root folder or create a new folder and add an `index.md` file to it. You can use any templating language supported by Eleventy. We use Markup, HTML, Nunjucks.

Page Front Matter

The page front matter works the same way as the post front matter. The only difference is that the `layout` property is required.

Example: file `features.html`

```
---
layout: default
title: Features
description: 'Stride supports an extensive list of features: Scene Editor, Physically Based Ren
# permlink is automatically generated based on the file name, but you can override it here
permalink: /my-features/ # otherwise it would be /features/
---
```

Shortcodes and Includes

To enhance the quality and functionality of the content, both pages and posts can incorporate shortcodes and includes. These tools offer a versatile way to enrich the presentation and interactivity of the content on the Stride website.

Web Assets

Our main web assets are:

- `css/custom-bootstrap.scss` - Slightly modified Bootstrap theme
  - Some Bootstrap variables are overridden
  - Some Bootstrap parts are disabled so they don't bloat the website (e.g. button-group, breadcrumb, ..)

- `css/styles.scss` - Main stylesheet
  - Styles also Dark Mode
- `css/syntax-highlighting.scss` - Imported prismjs styling, Light and Dark Mode
- `assets/search.liquid` - Script for search
- `assets/site.liquid` - Not used
- `assets/theme-selector.liquid` - Script for Ligth and Dark Mode selection
- `search.liquid` - Renders as `search.json` contains search meta

Styling

Bootstrap Customization

Our website uses the Bootstrap framework, version **5.3**.

> **IMPORTANT**
>
> Prioritize the use of Bootstrap's inherent styling before integrating any custom styles. You should be familiar with Bootstrap Utilities which help you to achieve most of the styling requirements.

CSS Guidelines

Our goal is to write as little CSS as possible to ensure the website remains lightweight. We maximize the utilization of the Bootstrap framework to achieve this.

Further, we are using also FontAwesome free icons. The icons are loaded in the `src/_includes/css/main.css` file.

Submitting your Changes

Assuming you have made all necessary changes and tested them on the development server, you can submit a pull request to the `master` branch. The pull request will be reviewed and merged by the website maintainers.

Steps to contribute your updates:

1. Commit your changes to your forked repository:
   - Commit the changes with a meaningful message
   - Push the changes to your forked repository
2. Create a pull request to the main repository:
   - You can create a pull request from your forked repository by navigating to Pull requests page and click **New pull request** button
   - Select the **master** branch as the base branch and your branch as the compare branch
   - Click **Create pull request** button

Once your pull request has been reviewed and approved, your changes will be merged into the main repository and deployed to the website.

Shortcodes and Includes

You can see examples here https://www.stride3d.net/blog/examples/.

Alert

To add an alert, use the following `include`, where:

- `type` is one of the following: `primary`, `secondary`, `success`, `danger`, `warning`, `info`, `light`, `dark`. Using these types will automatically include a relevant icon
- `icon` is a Font Awesome icon, which is optional. You can use any free icon, e.g., fa-check.
- `title` is the title of the alert

```
# This will render as a green box without the icon
{% include _alert.html type:'success' icon:'' title:'No icon: Stride contributors are proud to a

# This will render as a green box with a check icon
{% include _alert.html type:'success' title:'No icon: Stride contributors are proud to announce a

# This will render as a green box with a custom icon
{% include _alert.html type:'success' icon:'fa-face-smile' title:'No icon: Stride contributors a
```

Examples

See the examples here.

No icon: Stride contributors are proud to announce a new release now running on .NET 6 supporting the latest C# 10.

Custom icon: Stride contributors are proud to announce a new release now running on .NET 6 supporting the latest C# 10.

Default icon: Stride contributors are proud to announce a new release now running on .NET 6 supporting the latest C# 10.

Default icon: Stride contributors are proud to announce a new release now running on .NET 6 supporting the latest C# 10.

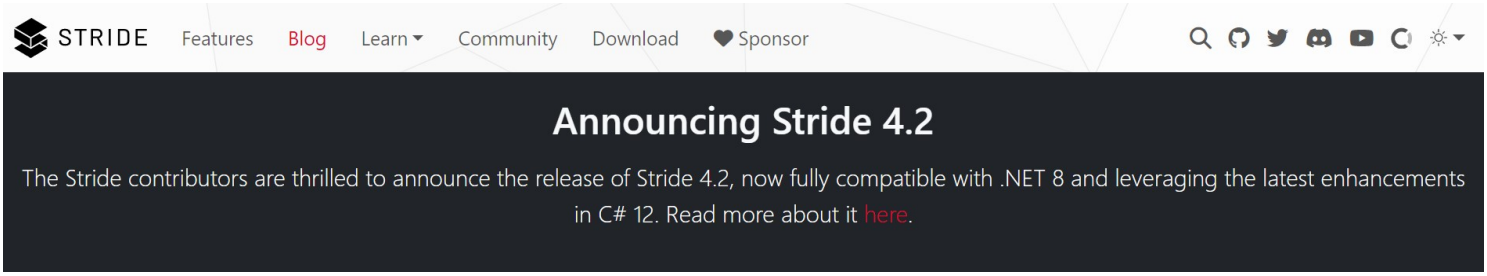
Default icon: Stride contributors are proud to announce a new release now running on .NET 6 supporting the latest C# 10.

Alert Banner

A global alert banner can be used for promotional purposes. The banner can be activated in `site.json`. It will show up on every single page.

```
"alert-banner": true
```

The HTML can be updated in the `/_includes/alert-banner.html` file.



Image

Add responsive images using shortcodes. Be sure to include a descriptive title, as it will improve your post's search engine visibility. Also, if possible, use the **webp** format for images, which can also be used for transparent images. This will improve the performance of your site.

img

To add a responsive image, use the following shortcode:

```
{% img 'title' 'url' %}
```

Replace `title` with a descriptive title for the image and `url` with the image URL. This shortcode renders as:

```
<img alt="title" src="url" class="img-fluid mb-2" loading="lazy" data-src="url">
```

img-click

To add a responsive image with a clickable link that opens the image in full size, use the following shortcode:

```
{% img-click 'title' 'url' %}
```

Replace `title` with a descriptive title for the image and `url` with the image URL. This shortcode renders as:

```
<a href="url" title="title" class="mb-2"><img alt="title" src="url" class="img-fluid" loading="
```

To add a responsive image with a clickable link that directs users to a custom destination, use the following shortcode:

```
{% img-click 'title' 'url' 'destinationUrl' %}
```

Replace `title` with a descriptive title for the image, `url` with the image URL, and `destinationUrl` with the target URL when the image is clicked. This shortcode renders as:

```
<a href="destinationUrl" title="title" class="mb-2"><img alt="title" src="url" class="img-fluid
```

Video

We should consider hosting our videos on YouTube whenever possible.

youtube

To embed a **YouTube video**, use the following shortcode:

```
{% youtube 'id' %}
```

Replace `id` with the YouTube video ID. This shortcode renders as:

```
<div class="ratio ratio-16x9 mb-2"><iframe src="https://www.youtube.com/embed/id" title="YouTub
```

youtube-playlist

To embed a **YouTube playlist**, use the following shortcode:

```
{% youtube-playlist 'id' %}
```

Replace `id` with the YouTube playlist ID. This shortcode renders as:

```
<div class="ratio ratio-16x9 mb-2"><iframe src="https://www.youtube.com/embed/videoseries?list=
```

To embed a video hosted elsewhere, use the following shortcode:

Hosting our own videos

```
{% video 'url' %}
```

Replace `url` with the video URL (e.g., .mp4 file). Make sure you have a matching .jpg file with the same name as the .mp4 file for the poster attribute. This shortcode renders as:

```
<!-- jpgUrl = url.replace(".mp4", ".jpg") // make sure you have a pair .mp4 and .jpg -->
<div class="ratio ratio-16x9 mb-2"><video autoplay loop class="responsive-video" poster="jpgUr
```

How to encode videos

Videos can be generated by many software in various formats & size, so they might end up being incompatible with web browsers or mobile, or simply be way too large. It is better to stick to a format with low requirements such as H264 baseline profile (works almost everywhere).

To do so, process the file using fmpeg:

```
ffmpeg -i myvideo_original.mp4 -profile:v baseline -level 3.0 -an myvideo.mp4
```

Also, generate a static thumbnail so that people can preview it before downloading the video (very important on mobile):

ToDo: Check if webp can be generated from ffmpeg

```
ffmpeg -i myvideo.mp4 -vframes 1 -f image2 -y myvideo.jpg
```

ToDo: Maybe we could provide a simple tool to do that without using command line.

Figma Designs

Stride boasts a range of official logos tailored for various applications and occasions.

Access the official Stride Figma designs to explore and utilize these creative resources.

Our Figma design collection encompasses:

- **Stride Logo**: The core visual representation of the Stride brand
- **Stride Icons**: A variety of icons reflecting Stride's identity and functionality
- **Stride Website Mockups**: Conceptual designs and layouts for the Stride website
- **Stride Tutorial Thumbnails**: Engaging and informative thumbnails for Stride tutorials
- **Stride Splash Screens**: Visually striking splash screens for Stride software and applications

Website roadmap

This document outlines a proposed roadmap and an ongoing development plan for our Stride website.

- **Address Existing Issues**: Prioritize resolving issues listed in the Issues section on GitHub.
- **Image Optimization**: Convert existing images to the WebP format to enhance website performance.
- **Decoupling Media**: Streamline the website by decoupling media from the site
  - Consider hosting videos on YouTube
  - Consider hosting images in Azure Blob Storage or another location

Eleventy site generator

Eleventy is a static site generator that uses JavaScript as its templating language. It is a very powerful tool that allows us to create a website with a lot of flexibility and customization. It is also very easy to use and learn. This section will cover the basics of Eleventy configuration on the Stride website. Creating and updating the content is described in our Content section.

We used to use **Jekyll** as our static site generator, but we decided to switch to Eleventy because of its flexibility and ease of use. We also wanted to use a tool that is more widely used and supported, which is why we decided to switch to Eleventy.

Packages and Dependencies

Eleventy is a **Node.js** application. Please follow our Installation guide to install Node.js and all the required dependencies.

Packages we currently use:

- Dev Dependencies
  - `@11ty/eleventy` v2.0 - Main package for the static site generator
  - `@11ty/eleventy-plugin-rss` - RSS feed plugin
  - `@11ty/eleventy-plugin-syntaxhighlight` - Syntax highlighting plugin (dark and light theme in `/css/syntax-highlighting.scss`)
- Dependencies
  - `@11ty/eleventy-fetch` - Fetch plugin
  - `@fortawesome/fontawesome-free` - Font Awesome with a variety of awesome icons 😃😻
  - `bootstrap` - Bootstrap 5.3
  - `lunr` - Lunr search plugin that consumes local `search.json (/search.liquid)` and remote `index.json` from the docs website; the script is in `/assets/scripts/search.liquid`
  - `markdown-it-anchor` - Anchor plugin for markdown-it
  - `markdown-it-table-of-contents` - Table of contents plugin for markdown-it, used mainly in blog posts as `[[TOC]]`
  - `sass` - Sass compiler for our `/css/*.scss` files

Configuration

The Eleventy configuration is located in the `.eleventy.js` file at the root of the project. This file contains all the configuration settings for the Eleventy build process. As it is a JavaScript file, you can utilize all JavaScript features and syntax within it.

What do you find in this file?

- plugins configuration - All the plugins we use

- pass through files - Files that are copied to the output folder without any processing
- custom collections - Custom collections used in the templates like `tagList` and `yearList`
- filters - Custom filters used in the templates
- custom shortcodes - Custom shortcodes used in the templates, pages or blog posts.

The file is well-commented and should be self-explanatory. If you need to add a new configuration, please follow the existing structure and include a comment to explain the new configuration.

Global Data

Global data is located in the `_data` folder. It contains all the global data that is accessible in all the templates. Currently, we have these JSON files:

- `site.json` - Contains all the global data for the website, used in the templates and shortcodes.
- `features.json` - Contains all the data for the features page and its features sections.
- `sponsors.json` - Contains sponsor information used in multiple places on the website.

Our `site.json` file contains these main properties, with only some listed below:

- `dark-mode` - Dark mode toggle `true|false`
- `alert-banner` - Global banner below navigation `true|false`
- `docs-search` - Includes docs website content in the search `true|false`
- `links` - Contains all the main links used across the website (social media, docs, GitHub, etc.)
- `authors` - Contains all the authors used in the blog posts
- repeated data - like `csharp-version`, `dotnet-version`, `download-version` which are used in multiple places on the website and are updated with every release

Folder Structure

The folder structure is crucial for Eleventy, as it determines the output of the build process. The folder structure is organized as follows:

Folders

- `_data` - Global data
- `_drafts` - Draft blog posts (excluded from the build process)
- `_includes` - Reusable code snippets that can be included in multiple pages
- `_layouts` - Main layout pages (`container`, `page`, `post`) using the primary layout page `default`
- `_site` - Output build folder (excluded in `.gitignore` and used for deployment)
- `assets` - Additional assets, such as scripts
- `blog` - Blog content page
- `css` - Website stylesheets
- `files` - Stride installer files
- `images` - Images and MP4 files used on the website

- `legal` - Content page
- `posts` - Blog posts
- `posts/2014-2021` - Old blog posts which are merged to the same output folder as `/posts`
  - this folder is only for convenience to easily access new posts
- `wiki` - GitHub wiki content - Excluded from build process, used only for wiki deployment. This will be decommissioned because the content was moved to Stride Docs

Files

- `posts/posts.json` - Blog post defaults so they don't have to be repeated in the front matter
- `*.html` - HTML content pages
- `*.liquid` - Liquid content pages
- `*.md` - Markdown content pages
- `*.njk` - Nunjucks content pages
- `.eleventy.js` - Eleventy configuration file
- `.eleventyignore` - Lists files and folders not to be processed by Eleventy
- `package.json` - Eleventy project metadata used by `npm`

Non Eleventy files

- `.nojekyll` - Special file for GitHub Pages
- `CNAME` - Custom domain for GitHub Pages
- `appsettings.json` - ASP.NET Core configuration file
- `appsettings.Development.json` - ASP.NET Core configuration file
- `Program.cs` - ASP.NET Core startup file
- `Stride.Web.csproj` - ASP.NET Core project file
- `Stride.Web.sln` - ASP.NET Core solution file
- `Stride.Web.csproj.user` - ASP.NET Core project file
- `web.config` - Configuration file for IIS deployment
- `web.Release.config` - Configuration file for Windows ASP.NET Core deployment

> **NOTE**
>
> This project includes ASP.NET Core solution and files, as they can be used seamlessly with Eleventy. Read more about this in our Installation section.

Layouts

All the layouts are located in the `_layouts` folder. The `default` layout is the main layout page and is used by all the other layouts.

- `default` - Main layout page

- `container` - Used by some pages
- `page` - Used by most of the pages
- `post` - Used by blog posts

Includes

All the includes are located in the `_includes` folder. The includes are reusable code snippets that can be included in multiple pages.

Some includes are used solely by the layouts, while others are used by the content pages.

Advanced Topics

Creating Custom Shortcodes and Includes

If you need to create a custom shortcode or include, please follow the existing structure and include a comment to explain the new shortcode or include.

The shortcodes are defined in the `.eleventy.js` file, while the includes are located in the `_includes` folder.

You can explore the existing shortcodes and includes to get a better understanding of how they work and how to create new ones.

Performance Optimization

ToDo: Remove this section if not needed

Deployment

Our team has explored various deployment options, ultimately selecting the method detailed in this guide for its efficacy. Additionally, for demonstration purposes, you can refer to the Deployment to GitHub Pages section for alternative deployment strategies you can use to showcase your updates.

Deploying to Azure Web Apps (Windows) with IIS

This guide is crafted for individuals who already have access to the Azure subscription. It provides step-by-step instructions for setting up a new Azure Web App, specifically tailored for staging environments. Note that the process for setting up a production environment is similar, but requires a distinct web app name.

Deployments to Azure Web Apps are automated through GitHub Actions, forming an integral part of our Continuous Integration/Continuous Deployment (CI/CD) process. The CI/CD pipeline is configured to automatically trigger deployments upon merging changes into either the `staging` or `release` branches.

> **NOTE**
> The deployment process outlined here is already established and running, hosted on Azure and sponsored by the .NET Foundation. This guide serves primarily as a reference for maintainers in the event that a new deployment setup is required.

Setting up a new Azure Web App

Follow these instructions carefully to establish your Azure Web App in a staging environment. For deploying in a production environment, replicate these steps with an alternate web app name for differentiation.

1. Navigate to the Azure Portal
2. Select **Create a resource**
3. Choose **Create a Web App**
4. In the Basic Tab
    - Choose your existing subscription and resource group
    - Under Instance Details, enter:
        - Name: **stride-website-staging**
        - Publish: **Code**
        - Runtime stack: **ASP.NET V4.8**
        - OS: **Windows**
        - Region: as the current web

        - Pricing Plan - An existing App Service Plan should appear if the region and resource group match that of the existing web app. Currently we use **Standard S1**.
        - Click **Next**
5. In the Deployment Tab - This step can be completed later if preferred.
    - Enable Continuous deployment
    - Select account, organisation `Stride`, repository `stride-website` and branch `staging`
    - Click **Next**
6. In the Monitoring Tab
    - Leave all settings as default
    - Click **Next**
7. Monitoring Tab
    - Disable Application Insights - This is not needed at this stage
    - Click **Next**
8. In the Tags Tab
    - Leave this blank unless you wish to add tags
    - Click **Next**
9. In the Review Tab
    - Review your settings
    - Click **Create**
    - The GitHub Action will be added to the repository and run automatically. It will fail at this stage, but this will be resolved in the subsequent steps.

> ⊗ **CAUTION**
>
> If you have completed the **Deployment Tab** process, ensure that the deployment profile includes the **DeleteExistingFiles** property. This property may need to be set to `False` or `True` depending on the specific requirements of your deployment. For instance, Stride Docs deployment retains files from previous deployments, allowing multiple versions like `4.2`, `4.1`, etc., to be maintained. Adjust this setting based on your deployment needs.

Adjusting the Web App Configuration

1. Proceed to the newly created Web App
2. Click on **Configuration**
3. Select **General Settings**
4. Change the `Http version` to **2.0**
5. Change `Ftp state` to **FTPS only**
6. Change `HTTPS Only` to **On**
7. Click **Save** to apply the changes

Modifying the GitHub Action

The previous step will have added a GitHub Action to your repository, which might fail initially. To address this, you need to modify the GitHub Action:

1. Navigate to the repository
2. Select **Actions**
3. You have the option to stop the currently running action
4. Locate the new GitHub Action file *(stride-website/blob/master/.github/workflows/some-file-name.yml)* that was automatically generated by Azure Portal. We need to extract the `app-name` and `publish-profile` values from it and disable the push trigger.
   - To disable the push trigger, retain only **workflow_dispatch** (manual trigger) as shown below:

```
on:
#  push:
#    branches:
#      - staging
    workflow_dispatch:
```

5. Open the `stride-website-staging-azure.yml` workflow and update it with the values obtained in the previous step. Save your changes.
6. This workflow might also need to be added to the `master` branch if it is not already present.
7. Execute the workflow `stride-website-staging-azure.yml`. Ensure you select the correct branch `staging` and click **Run workflow**. This action will deploy the website to the Azure Web App.

GitHub Actions

- `stride-website-github.yml`: Facilitates manual deployment to GitHub Pages in the forked repository, primarily used for showcasing updates
- `stride-website-release-azure.yml`: Automates deployment to production upon merging changes into `release` branch, with a manual trigger option also available
- `stride-website-staging-azure.yml`: Enables automatic deployment to staging upon merging changes into `staging` branch, along with an option for manual triggering
- `stride-website-wiki.yml`: Automatically deploys to the GitHub Wiki when changes are pushed to the `wiki` folder in the `master` branch, also includes a manual trigger feature

Deployment to GitHub Pages

To showcase your updates, especially helpful for design changes pending review, you can deploy the website either to your infrastructure or to GitHub Pages, a free hosting service. Once deployed, share the link with us for review.

Prerequisites

In your `stride-website` repository:

1. Navigate to **Settings** → **Actions** → **General** → **Workflow Permissions**
    - Choose **Read and write permissions**

Run GitHub Action

1. Go to **Actions**, select **Build Stride Web for GitHub Staging**
    - Click **Run workflow**; you may optionally select a branch
2. Monitor the build logs while the action is in progress
3. Upon successful build, a `gh-pages` branch will be created
4. Navigate to **Settings** → **Pages**
    - Choose the `gh-pages` branch with the root option and click **Save**
5. After saving, an internal GitHub Action **pages build and deployment** is automatically created and triggered, deploying the content to the GitHub Pages website
6. Initially, the website will be accessible at `https://[your-username].github.io/stride-website` but with broken styling

Add Custom Domain

1. To resolve styling issues, deploy the site to a custom domain. This is necessary because the site isn't deployed at the root directory on GitHub Pages
2. Go to **Settings** → **Pages** → **Custom domain**
    - Enter your custom domain and follow the instructions for verification
3. Upon saving, the **pages build and deployment** action is triggered again, adding a `CNAME` file containing your custom domain name to the `gh-pages` branch
4. Your website should now be fully operational on your custom domain, for example, `https://stride-website.vaclavelias.com/` is hosted on GitHub Pages

Major Release Workflow

1. **Create a Release Blog Post**
    - Place the post inside the `_drafts` folder, which is not deployed, or directly in the `posts` folder for testing in the `staging` environment. Note: If you need to deploy updates to the `release` branch, remember to move the post back to the `_drafts` folder.
2. **Update `_data\site.json` with the following settings**, which are used in multiple places on the website:
    - `version`: Increase the version number
        - This helps refresh the cached CSS file
    - `download-version`: Update the version number for downloads
        - Used on the home page
    - `csharp-version`: Update to the current C# version being used
        - Used on the home page and features page
    - `dotnet-version`: Update to the current .NET version being used
        - Used on the home page and features page

Troubleshooting and FAQ

Known Issues

1. **Sponsor Page - Widget Incompatibility with dark theme:** Widgets on the Sponsor Page currently do not support the dark theme. As a workaround, we can either fetch data from https://opencollective.com/stride3d/members/all.json and render it before deployment or make it dynamic. Both options would give us more control over the content and design, and allow for better compatibility with the dark theme in the future.
2. **Search Page - Lack of pagination:** At present, the Search Page does not have pagination, which limits the maximum number of search results displayed to 100. To resolve this issue, we can implement a pager in JavaScript. This would enable users to navigate through multiple pages of search results, providing a more comprehensive view of the available content.

Common Issues and Solutions

Any issue should be added to Stride Website GitHub issues so it can be tracked and elaborated by the community.

Frequently Asked Questions

Q: I just want to fix a typo in a post. Do I need to follow your installation steps?

A: *No, you can fix the typo directly on the GitHub website. However, you will still need to fork the repo, make your update on the main branch or a new branch, and then create a pull request. You can follow this guide for minor updates.*