# BuildStream Architectural Overview (v2.2)

## What is BuildStream ?

BuildStream is a flexible and extensible framework for the modelling of build and CI pipelines in a declarative YAML format, written in python.

BuildStream defines a pipeline as abstract elements related by their dependencies, and stacks to conveniently group dependencies together. Basic element types for importing SDKs in the form of tarballs or ostree checkouts, building software components and exporting SDKs or deploying bootable filesystem images will be included in BuildStream, but it is expected that projects forge their own custom elements for doing more elaborate things such as running custom CI tests or deploying software in special ways.

The build pipeline is a sort of flow based concept which operates on filesystem data as input and output. An element's input is the sum of it's dependencies, sources and configuration loaded from the YAML, while the output is something on the filesystem which another element can then depend on.

## Project Requirements

- **Limited Scope and Maximum Extensibility**. By providing limited built-in *Element* implementations, we externalize the problem of supporting every deployment (or *export*) method under the sun. By allowing users to implement their own *Elements*, we are best prepared to handle use cases that we did not foresee.

- **Backwards Compatibility**. By providing an API surface that is limited and stable, and restricting how the underlying YAML format changes, we ensure as a rule that future versions of BuildStream remain usable with older projects, even when those projects provide custom *Elements*.

- **Limited scope of builds**. Instead of mandating that an entire operating system be bootstrapped and built from the ground up, we allow building a pipeline on top of a different base, or on top of another pipeline output, using the concept of *imports*. An *import* need only be an *Element* which creates an artifact from an abstract source:
  - A tarball SDK (such as KC sysroots delivered by Intel or any other binary blob SDK)
  - A flatpak SDK (essentially a signed OSTree commit)
  - An artifact output of a separate build pipeline (allowing for recursive building of pipelines)

- **Project Modularity**. Projects can be combined with one another by way of recursive pipelines. When building an entire OS, one should be combining multiple projects. Downstream projects have control of exactly what version of stacks they consume. Hosting all build instructions for an entire OS in the same git repository is not encouraged.

- **Build Configuration**. A default configuration is shipped with BuildStream, defining sane defaults for variables such as *prefix*, *execprefix*, *sysconfdir*, *localstatedir*, etc. Commands for configuration used by various build systems are also defined using these defaults. A BuildStream project configuration file may override some of these values so that the entire pipeline is built differently.

- **Developer Convenience**. It should be possible to integrate workspaces into the build, so that developers may make stack-wide modifications and build and test the results *before* having to stage any commits. We should also be able to easily shell into a build environment and run, debug, manually build programs and run them at will.

- **Varied Build Planning Options**. This concerns the decision that BuildStream makes about what needs to be rebuilt or rebuilt. In the usual context the build is deterministic, cache keys determine what needs to be rebuilt and every element which depends on a changed element needs to be rebuilt. In the non-deterministic case, we need a build plan which only rebuilds elements who’s cache key has changed since the last invocation, but not elements which depend on that element (This is in consideration of GNOME Continuous CI).

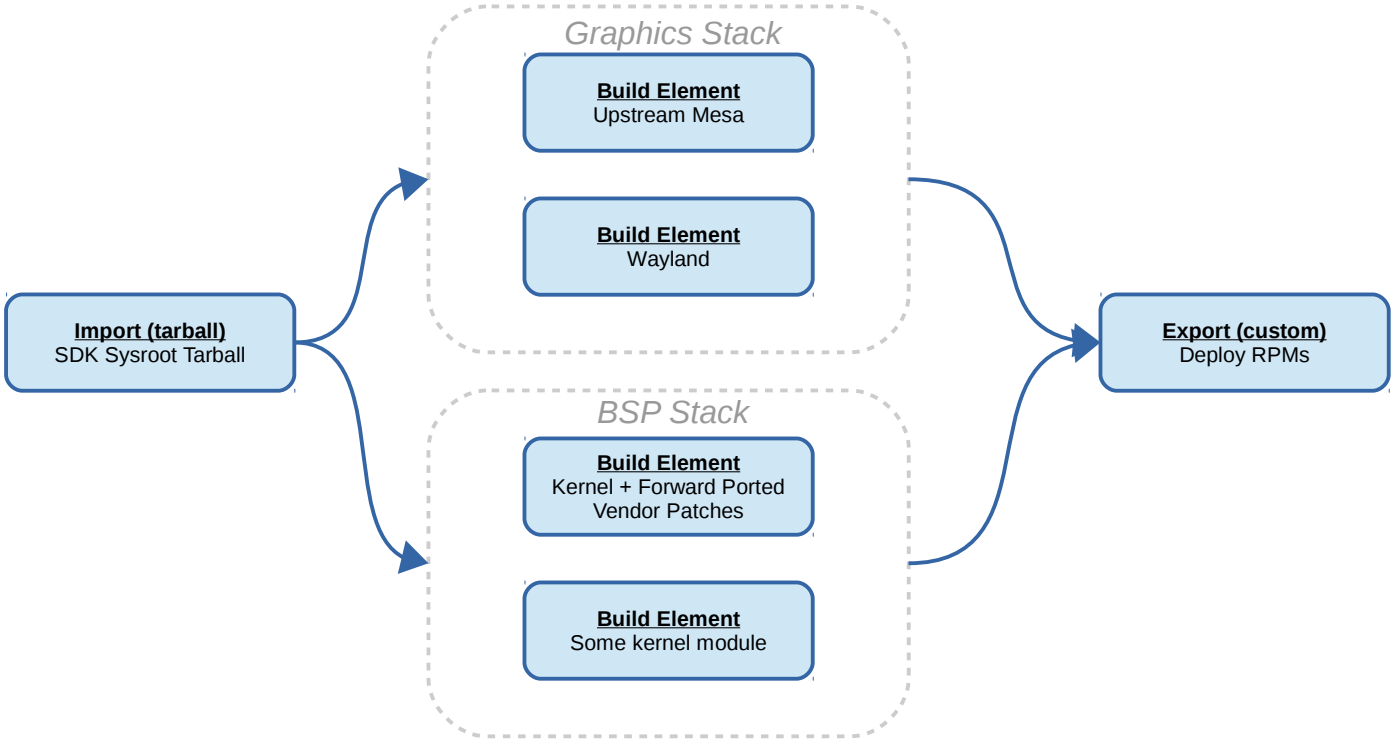
- **Deterministic, Repeatable and Reproducible Builds**. Not entirely in the scope of BuildStream itself but something we make possible. Host tooling is only ever used to obtain sources or by a BuildStream project which bootstraps a base runtime, after this point only tools from within the sandbox can be used. Repeatability is a matter of ensuring one relies on minimal standard tools for the bootstrap process. Reproducibility is considered by BuildStream when collecting build output (file mtime) and when implementing the Sandbox, but is also sometimes a matter of how the sources are actually written.

The YAML format is a method for the modeling of *Pipelines*, a *Pipeline* is a collection of *Elements* which can be assembled and deployed. *Elements* in a pipeline are related by depending on previous elements in the pipeline. While all elements implement the same abstract interface, for practical purposes we can assert that elements fall into the following basic categories:

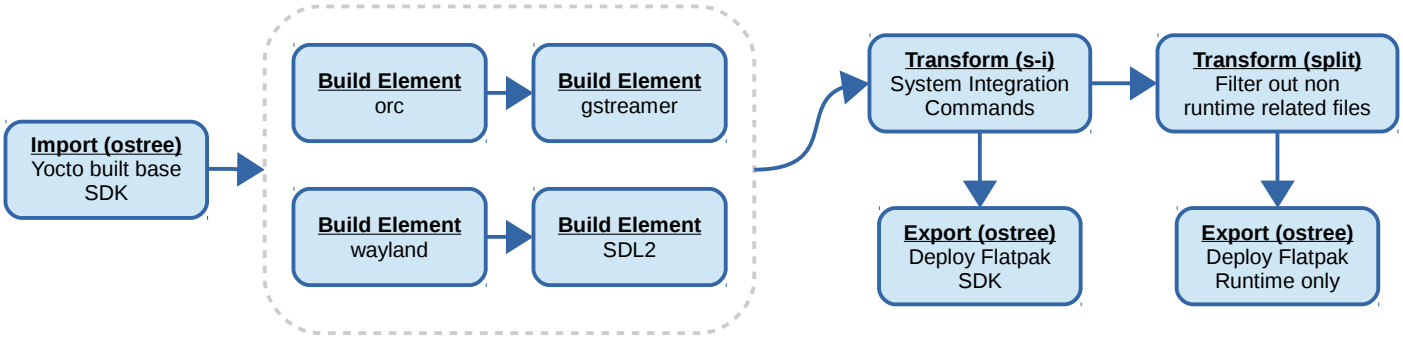
- **Build Element**: Runs tooling provided by dependencies in the sandbox to produce an artifact
- **Stack Element**: Depends on build elements which it *contains*, all of which depend on the group element's dependencies themselves (this element is treated as a special case)
- **Import Element**: Imports some external source, converting it into a local artifact
- **Export Element**: Turns it's dependencies into something useful, like a bootable system image
- **Transform Element**: Processes it's dependencies in some way to produce a new transformed artifact, such as running system integration commands or filtering by artifact splitting rules

# Example Visualizations of Plausible Build Pipelines

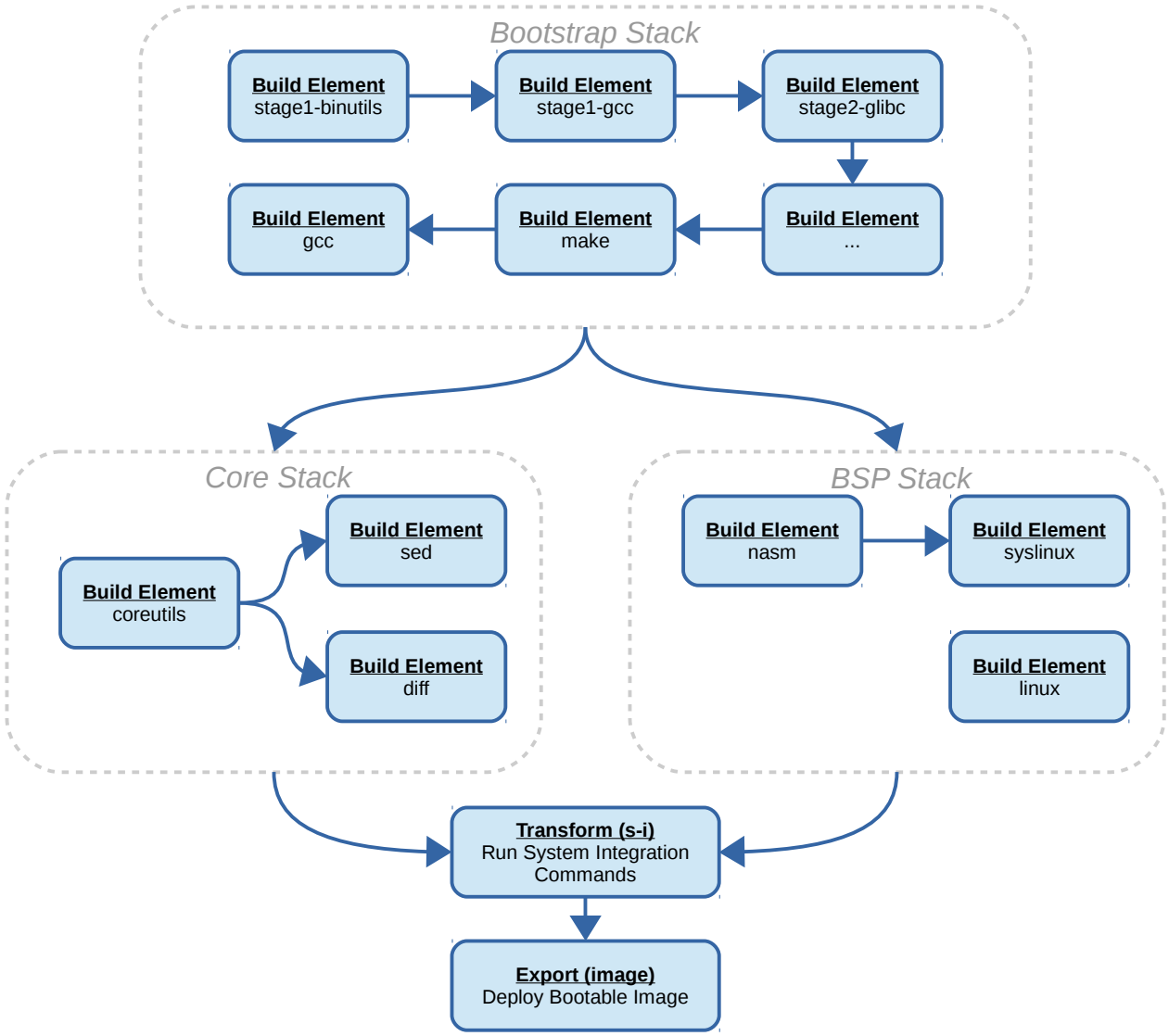
*Build some things on top of a sysroot tarball and deploy some rpms*



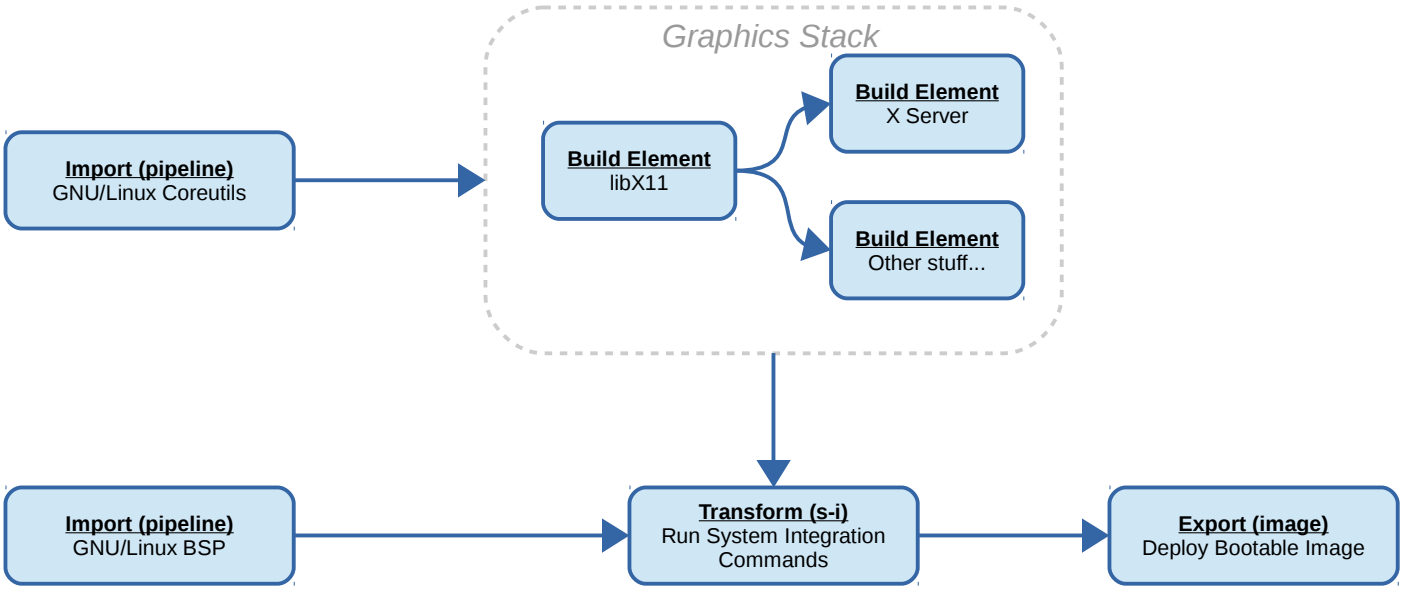

*Build Flatpak SDK or Runtime on top of another Flatpak SDK*



*Build a minimal bootable base linux system*




*Build an extended bootable linux system*

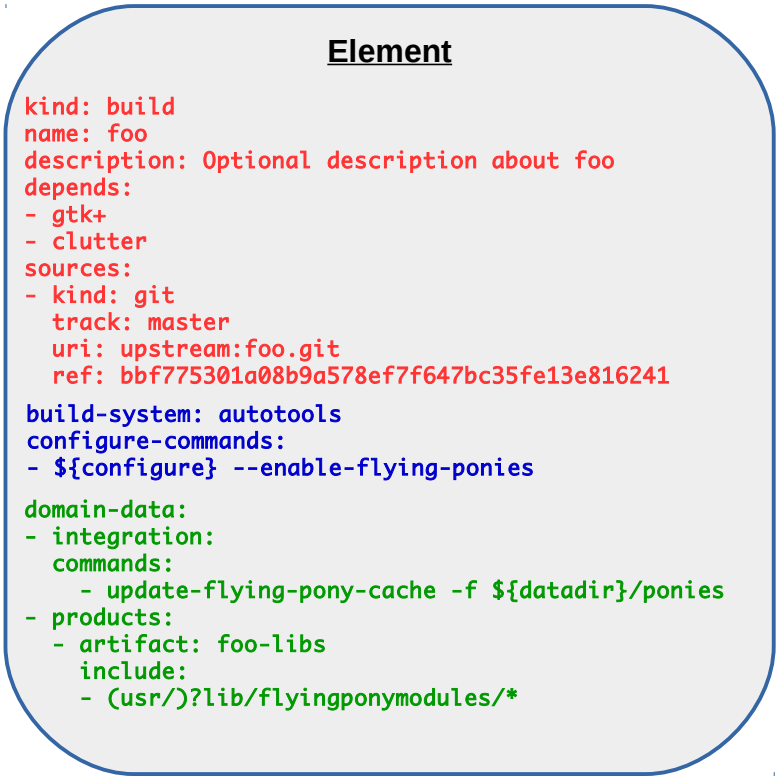


# Elements

This abstract element is the root of all evil in the Build Stream architecture, every element must conform to this interface, custom elements may be written to implement exotic elements.

The concrete type used to construct an element is depicted by the `kind` attribute specified in the YAML for the specified element. All data specified for the given entity in the YAML (except for *sources* and dependency specifications) is given to the element instance as a dictionary and is considered to be exclusively in the domain of the given element type. Dependency information remains in the domain of the *Pipeline* itself. *Sources*, which are abstract objects of themselves, are constructed separately and handed over to the Element separately from the rest of the dictionary. Additional public *domain data* may be assigned to any Element. An Element can access the domain data of all of the elements on which it depends at assembly time.

## Anatomy of Element Data




### Pipeline Private Data

The pipeline creates the abstract *Source* objects and hands over the name, description and *Source* objects to the *Element* at construction time, the *Element* type is derived from the specified *kind*. Knowledge of how dependencies are expressed is in the domain of the pipeline, while name, description are public, and build sources are shared with the element itself.

### Element Configuration Data

The element is in control of it's own configuration data which is private for a given element. This is used to configure how an element operates on it's inputs to create it's output.

### Public Domain Data

Extra data can be declared on an element, this data is visible to the element itself, and all elements which depend on the element later in the pipeline. This can be useful for custom elements to consume later, or for the same element type to consume from other instances.

This layout is chosen with API stability and extensibility in mind:

- The dependency information belongs strictly to the pipeline core which is responsible for parsing the YAML initially and constructing the dependency tree of Elements. This allows some measure of freedom in how dependency information can be expressed in future versions of the format.

- Element private data is defined exclusively by the element itself. First class citizen element formats may evolve with revisions of the main BuildStream format, but an element must continue to understand and be backwards compatible with older formats.

- Public data categorized by domain names allow elements to read domain specific data on all the elements which they depend. This also allows developers of custom elements to add extra annotations across the entire pipeline which can be processed by their elements.

## Default Configuration Data

When defining an *Element* type, the element will provide a class-wide dict which expresses the default *Element Configuration Data* (as defined on the previous page). This means that in a user project, what the user specifies in YAML is not the sum total of the data given to the *Element* at instantiation time. Instead, the dictionary used to configure an element is composed in three stages:

1) First a copy of the Element class-wide default dictionary is made.
2) The project configuration is read, which can override the default data of a given element type by composing a similar dictionary.
3) Finally the dictionary parsed from the *Element* declaration YAML is used to override the above.

## Element Base Class Methods

Some basic knowledge about Element inputs and outputs are handled by the base Element class from which all Element implementations derive. These methods are generally used by Element plugins at *assemble* time but also useful for general pipeline operations.

- `dict = element→domain_data (domain)`

  Fetches the public *domain data* on the element for the specified *domain*.

- `element→stage (sandbox, directory=None)`

  Stages the element itself in the sandbox, by consulting the artifact cache for an artifact with the correct cache key for the given element and asking the sandbox to stage it. If directory is specified, stage to that sandbox relative directory, otherwise assume “/”.

- `element→stage_dependencies (sandbox, directory=None)`

  Stages the dependencies of the element, by simply walking the dependency tree from the base of the pipeline up to the element, not including the element, and calling the *stage* method on each of these elements in the correct order.

- `element→stage_sources (sandbox, directory)`

  Stages the element’s sources in the sandbox, by iterating over the element’s sources and asking the sandbox to stage them.

- `element→integrate (sandbox)`

  Runs integration commands for the element on the sandbox. By fetching the “integration” domain data and running the commands specified.

- `element→integrate_dependencies (sandbox)`

  Like *element→stage_dependencies()*, but runs the integrate method.

# Element Abstract Methods

Below is a draft of what the Element's abstract methods will probably look like:

- `new Element (dict, sources[])`

  The element constructor is given its dictionary which is parsed from the YAML, along with 0 or more *Source* objects. The base element class initializer takes care of holding a reference to the passed *Source* delegate objects so that they are attached and available at any time.

- `cacheKey = element→enrichCacheKey (cacheKey)`

  The element is responsible for tracking what parts of it's configuration data may effect the build output, and also uses it's Source delegates to assist in enriching a cache key. The passed cacheKey is at first the context of the build itself (target architecture, build environment variables, and anything else) and then an accumulation of the given element's dependencies.

- `directory = element→assemble (sandbox)`

  The juice of the element implementation is a matter of producing some output, the element is given an initialized *sandbox* object and the element may then use the *stage_dependencies()* and *stage_sources()* methods to prepare the sandbox.

  Note that some deployment related elements may choose to stage the dependencies of a different target at the root and stage it's own dependencies in a subdirectory, this is to ensure that we may always use tooling that we've built and never rely on host tooling to perform a deployment or "export".

  Here the element should interpret it's configuration data and operate on the prepared sandbox to produce output. Normally the element will use *sandbox→run()* to run programs within the sandbox in order to create the output.

  After output is generated inside the given sandbox, the element returns the sandbox relative path indicating what should be the root of the produced artifact. This is typically "`/${element_name}.inst`" for a *Build Element* and "*/*" for a *Transform Element*.

# Sources

Similar to the abstract *Element* class, the *Source* is an abstract interface who's type is indicated by the `kind` attribute in YAML and for which there will be some built-in implementations but third parties are allowed to provide their own *Source* implementations in their own projects (although this is less expected).

The roles of a Source consists of:

- Obtaining files from some remote location (usually source code)
- Caching the sources locally
- Calculating a cache key for the given set of sources
- Allowing the sources to be staged into a sandbox
- Tracking of remote branches or updating an sha256 sum

## Anatomy of Source Data


### Pipeline Private Data

The pipeline recognizes the *kind* attribute at parse time and uses that to determine what type of *Source* to instantiate.

### Common Data

Every *Source* has the *stage* attribute in common, this can usually be omitted but indicates the staging relative directory where the source should be placed when staging multiple sources for a given *Element*. The jury is out on whether this needs to be public or not.

### Source Configuration Data

Like an *Element*, the *Source* is in control of defining what attributes are necessary to fetch the source files. In the case that the *Source* type represents a VCS of sorts, it should use a *track* parameter to indicate a symbolic name of a branch from which it can update it's own *ref*. As a matter of convention, Source implementations should use the words *track*, *uri* and *ref* if at all possible.

## Aliases

A brief note about aliases is in order. Note in the above YAML the word "upstream" is used instead of specifying the full URI for a given git repository. The Source object will use a utility function in the BuildStream core to resolve a full URI based on a URI which used such a prefix.

Aliases are configured in a BuildStream project root in a project wide project.conf file which will have a section for assigning base URI paths to symbolic aliases.

## Source Abstract Methods

This interface to implement for a *Source* object type is approximately:

- `new Source (dict)`

  The build source is given a dictionary obtained from the YAML indicating some parameters, such as a git repository and commit sha, or tarball url and sha256 sum, or whatever the given *Source* might need to obtain an exact set of sources.

- `cacheKey = source→enrichCacheKey (cacheKey)`

  Similar to the *Element*, and contributing to the *Element*'s cache key, the *Source* should provide a method for enriching a cache key with some information indicating exactly what input it provides to the Element.

- `source→fetch ()`

  Ensure that the required files are available in the local cache directory, possibly by mirroring a git repository, or by downloading a tarball or any other means. This would be a no-op for a *Source* implementation which stages files directly from the BuildStream project directory.

- `source→stage (directory)`

  Stage the sources at a given directory. Regardless of whether or not the sandbox in use needs to translate file attributes for sources to be recognized correctly in the build environment, an entry point is required for the source to prepare a directory somewhere.

  Most probably this will always be called with a directory owned by a given sandbox instance.

- `source→refresh (dict)`

  Refresh the source *ref* based on it's tracking branch, if possible.

  A *Source* for a tarball might omit the sha256 sum or a *Source* for a VCS might provide a only a named branch. This will result in a *Pipeline* which cannot be run, however a traversal of the *Pipeline* calling the *refresh* method on all associated *Source* objects will result in updating the parsed YAML inline so that the pipeline can then be run.

  Even if the *ref* is currently set in the parsed YAML, the *refresh* method can be used at any time to fetch the latest *ref* on a remote tracking branch in the case that the *Source* type represents a VCS.

- `source→assert_host ()`

  After the *Pipeline* has been constructed and before processing, an initial traversal of the pipeline is made to ensure that the build host is capable of obtaining sources. This method should assert the presence of git or svn or whatever host tooling is required to obtain sources according to the *Source* implementation, raising an informative exception if the required host tools are lacking.

# Pipeline

The *Pipeline* represents a parsed pipeline definition in YAML and is the owner of all elements in the pipeline, exposing some methods to obtain elements or to iterate over them in useful ways.

The knowledge of how *Elements* and *Stacks* come together by relation of their dependencies, and how *Elements* are included in *Stacks*, is entirely in the domain of the *Pipeline* object. In other words, moving forward; if and when the overall YAML format for expressing dependencies changes, the charge of understanding previous formats for expressing dependencies and such element relations lies on the *Pipeline*.

The pipeline is also the main outward facing API for running pipelines, as such it will have an entry point for running all the commands which the CLI frontend offers.

The user facing API is approximately:

- `new Pipeline (directory, target, arch)`

  Where *directory* is the toplevel directory of a given pipeline project, *target* is the *directory* relative path to the target element definition to build, and *arch* is the architecture to build for.

  Note that when the user invokes the build-stream program on the command line for a given activity, a pipeline will be created for the target every time. As such, the target of a pipeline remains in context for the lifetime of a given pipeline object.

- `pipeline->run ()`

  Assembles the elements in the pipeline. This may grow some additional arguments, such as whether the pipeline should be run interactively, or how to react to failure conditions.

- `pipeline->refresh ()`

  Iterates over all the Source objects associated to Elements in the pipeline and calls the refresh method on those sources. The pipeline must keep a cached version of the original parsed yaml dictionary and pass the Source yaml fragments to their respective sources here.

  After traversing the dependency tree, if no error is encountered, the resulting YAML is rewritten using a round-tripping YAML implementation such as ruamel.

- `pipeline->shell ()`

  Runs the Element's methods to stage the dependencies and run the integration commands of the pipeline target, and then runs a shell from the given sandbox. This method might take an argument to distinguish whether the target element itself should be staged or only it's dependencies.

  Error out if the target's dependencies have not yet been assembled.

# Project Configuration

We have made reference to a project wide configuration file several times now, lets take a brief look at what goes in here, although this is likely subject to changes and possibly will grow with time.

The purpose of a project configuration file is to define aspects particular to the given project, not the context in which possible pipelines in the given project will be run. User preferences do not go in this file.

- **Aliases**: The project itself makes use of aliases to define source URIs conveniently in shorthand, these are defined here.

- **Element Overrides**: The project configuration may include an optional section which defines overrides for the default dictionaries declared on Element class-wide data. This allows the author of a project to augment the default behavior of elements within that project.

- **Environment Overrides**: This allows one to augment the default *Sandbox* environment for the entire project.

# Element Implementations

Here we give a bit of a brief overview on the different element types to be implemented as first class citizens and how they work.

**Build**

The Build Element is the one who's dictionary consists of command lists such as *configure-commands*, *build-commands* and *install-commands.* These have defaults according to the specified *build-system* which is also a value in the domain of the Build element.

This element assembles it's artifact by staging all dependencies into the sandbox's root directory, staging the sources into a build directory and running the commands found in it's dictionary.

**Stack**

The Stack Element is treated as a special case by the *BuildPipeline*, when the pipeline encounters a *Stack* element, it causes every element which is *contained* in the group to also depend on the *Elements* or other *Stacks* which the given *Stack* depends on itself. The *Stack* itself then depends on it's contents in the underlying pipeline.

Asides from the matter of how dependencies are sorted, the *Stack* is mostly just a symbolic element which makes a statement that it's contents have been built, it's assemble implementation simply creates an artifact with some metadata describing it's content.

**ImageExport**

The ImageExport Element's dictionary specifies some information such as what filesystem type should be used and what size the resulting image should be, whether it should be a bootable image, etc. Additionally it must specify an element in the pipeline which guarantees the presence of the tooling required to deploy the image, utilities such as mkfs and syslinux.

To implement the export, the ImageExport element will first stage the required tooling element and it's dependencies in the root of the given Sandbox, and then it will stage it's own dependencies into a build subdirectory. The element will then proceed to run commands in the sandbox to deploy the build subdirectory as an image in the output directory.

**OSTreeExport**

Similar to the ImageExport, except that it will require staging of an element or group which includes ostree and will use the built ostree to deploy it's own dependencies to an output OSTree repository.

It may take a GPG signing key as one of it's parameters in it's own dictionary.

**IntegrationTransform**

The IntegrationTransform element will stage all of it's dependencies at the root of the given Sandbox and run system integration commands from each of it's dependencies, in order of dependency, directly in the root of the sandbox. The output of this Element is the root directory of the sandbox itself.

**RawImport**

This is an *import* type of Element which is used for constructing the base of a Pipeline, this one specifically is used for importing SDKs in tarball form to be built on top of, but we call it a *RawImport* since it is perfectly capable of importing anything for which there exists a *BuildSource* implementation.

The RawImport assemble implementation is performed simply by staging its BuildSource directly into the sandbox root directory and reporting the root directory as output, resulting in an artifact being created which contains the imported payload.

For the purpose of importing OSTree checkouts as a base to build on top of, a *BuildSource* should be implemented. OSTree imports can then use the same RawImport Element.

**PipelineImport**

This element provides a way for chaining multiple pipelines together. It will use an arbitrary *BuildSource* to stage a foreign BuildStream project and then instantiate a new *Pipeline* to build that project while passing any user preferences and invocation context, such as user configuration and target architecture.

It should be noted here that only one pipeline should ever be running at once in a given invocation context, when the invoking Pipeline recurses into a child Pipeline, the parent is effectively slaved to the child Pipeline and relinquishes control while the child pipeline is processing.

# The Sandbox

Up until here we have covered most if not all of the legwork. What remains is the Sandbox and the Artifact staging and collection, this still requires a bit more thought and investigation in order to ensure we get the minimal public API right the first time around.

Tentative API draft:

- `setupSandboxBackend (targetArch)`

  Depending on some criteria, like if we are cross compiling and have a VM ready to perform builds inside of, or whether we have tooling present to virtualize being root (such as fakeroot and or bubblewrap) the engine can decide which sandbox backend is appropriate.

  If compiling on the same arch, nothing special is needed, chroot with root permissions or bubblewrap will suffice. If compiling for a host compatible architecture, i.e. when targeting a 32bit arch on a 64bit host, then `linux32` or similar should also be employed automatically when running commands in the sandbox. If cross compiling, then a bootstrapped qemu image, kernel and cross compiler for distcc would be required, or a scratchbox2 environment might be an alternative.

- `new Sandbox (bootstrap=False)`

  Setup a new build directory on the host which can be accessed by the sandboxed environment, either by chrooting directly into it, or providing an nfs or virtfs mount point to share the directory.

  If the bootstrap option is specified, then the sandbox should be usable without a runtime, using host tooling to allow the user to at least run a shell. Depending on the Element which requested a bootstrap sandbox, some host tooling assertions should be made to ensure the required host tools are available for the Element to assemble itself.

- `sandbox→cleanup ()`

  Remove the content of the sandbox from existence.

- `sandbox→stage_source (source, relative_directory)`

  Stages the *Source* in the sandbox in such a way that the sandbox environment understands the content in the same way it appears inside the artifact (i.e. this must handle sandbox environment specific details, like when staging files on a share directory accessible in a VM with a virtfs mount).

  In most cases this can simply be implemented by requesting the *Source* to stage itself at a given path.

- `sandbox→stage_artifact (artifact_filename, relative_directory)`

  Similar to stage_source(), except that here we deal with an artifact. For convenience the sandbox may maintain a cache of extracted artifacts who's content can be hardlinked into the target sandbox. This operation must however respect the file attributes as recorded in the artifact tarball so that they appear the same when viewed from the sandbox, even if they may not appear the same way to the regular user on the host system.

- `sandbox→run (mode, cwd, environment[], command)`

  Runs *command* in the specified sandbox relative working directory *cwd* and with the specified *environment* variables set.

  The *mode* parameter here indicates what permissions should be given to the sandbox. This can either be *permissive*, with network accesses for the purpose of running and testing programs built inside the sandbox, or it can be *sealed*, which is what we want to use for deterministic builds.

- `sandbox→shell (mode, cwd, environment[])`

  In some sandbox environments, this may be equivalent to simply running an interactive shell with sandbox->run(). However this API is special because it must ensure that the calling terminal is connected to the sandbox environment. This will particularly require extra legwork when running a VM sandbox for a virtualized cross build.

The above *should* hopefully provide everything we need for operating within the sandbox. As we cannot assume the outer environment has permission to create device nodes; creating those must be implemented by running tools staged and accessible from within the sandbox (mknod). Also since we ditch the concept of “extensions” in favor of extensible build *Elements* which can themselves make use of a sandbox, we should not really have need for executing any commands in the host environment in a sandbox directory.

## Artifact Cache

The remaining piece of the puzzle is yet to fall in place, the Sandbox code itself clearly needs to be responsible for the creation of artifacts from a sandbox and for the staging of an artifact in the sandbox, and that is most of the trouble. What remains is only managing the artifacts and downloading them from a shared server so that not everything needs to be built on everyone’s host.

This may be a private utility inside the BuildStream package, and may be invoked by way of some common code on the general *Element* class, possibly the artifact can attempt to be downloaded before ever calling `element→assemble ()`, we’ll have to just see what is most practical, it should however remain a hidden implementation detail and not exposed in our APIs.

One thing worth consideration is killing away the kbas server entirely, and using an apache hosted OSTree repository instead to cache the artifacts. This would let us remove one more custom moving part in favor of something moderately widely used and equally easy to setup; with the added benefit that GPG keys can be used to sign officially built artifacts and verify them at download time for free.

NOTE: In order to use ostree in place of an artifact server, we would either need to have the artifact server pull from dedicated build machines in order to populate the cache (which *might* be a good idea, preventing random developers from uploading their own artifacts), or alternatively, we would need complete the ostree work to support the ostree push semantics.