Proposal for a "What-Goes-Where" Provider

Introduction

This proposal outlines the creation of a new, small Bazel module (or addition to an existing “base” module, like bazel_lib) designed to unify the definition of file placement within archives, container image layers, and other deployment targets. This "what-goes-where" provider aims to decouple the description of file contents (known by language rulesets like `rules_python`, `rules_js`, or bazel-lib’s `copy_to_directory`) from the specific implementation details of packaging or deployment rules (e.g., `rules_pkg`, `tar.bzl`, `rules_img`). This approach seeks to avoid issues similar to those encountered with `rules_docker` by providing a clear and extensible mechanism for specifying file locations and metadata in each language ruleset (and not in a single place shared by all languages).

The information we want to capture includes:

- Which `File` structs should end up in a deliverable (and which can be safely omitted)
    - This is important when bundling an executable file, where the `DefaultInfo` of the executable may reference a different subset of `File` than what we would like to bundle in a deliverable.
- How the deliverable can be split up into smaller components (like container image layers)
    - Language runtime, standard library, and similar foundational components
    - Third-party runtime dependencies (shared libraries, interpreted code, ...)
    - Application code or (same-party) runtime dependencies
- How the files should be laid out in the deliverable
    - Could reuse runfiles layout of an executable.
    - Sometimes, a different layout could be necessary.
- Additional filesystem nodes that aren’t well expressed by `File`
    - Empty directories that should be created in the deliverable
    - Symlinks that should be created in the deliverable
        - This is different from `File.is_symlink`: we care about the representation of symlinks as a pair of strings in the deliverable.
- Other metadata, including file attributes
    - mtime of files and other timestamps

- Ownership information
- Extended attributes (xattr)
- Filesystem flags (rwx + more)
- And potentially extra metadata that is specific to the kind of deliverable, like custom PAX headers in a tar file. This means we want to allow for custom metadata tags, along with a few well-known ones.

Problem Statement

Currently, various Bazel rulesets for packaging and deployment handle the mapping of source files to destination paths independently. This leads to redundant implementations and tight coupling between language rulesets and specific packaging solutions. A centralized, standardized provider for "what-goes-where" information is needed to:

- Enable language rulesets to describe archive or container image contents without depending on specific archive implementations.
- Decouple the "what-goes-where" description (known to language rulesets) from container image or archive rules, reducing complexity and increasing flexibility.
- Facilitate efficient deployment of files to object storage and other targets by providing a unified input format.The current approach often involves language rulesets independently creating entire tar files, leading to inefficiencies. For examples, see:
- `rules_js`: [js_image_layer]
- `rules_py`: [py_image_layer]

High-level interaction

A high-level interaction would involve a language ruleset (e.g., `rules_python`) producing a `what_goes_where` provider. A packaging rule (e.g., `rules_pkg`'s `pkg_tar`) would then consume this provider. For instance, a `py_binary` target could expose this provider, and a `pkg_tar` rule could use it to specify the contents and layout of the generated tar archive without the `py_binary` rule needing to know the specifics of tar file creation. This effectively separates the concern of "what files are needed" from "how those files are packaged."

Here's a rough example illustrating how a `py_binary` rule implementation could return the new `FSManifestInfo` provider (working title) and how an `image_layer` rule from `rules_img` might consume it in a `BUILD.bazel` file.

```python
# In a hypothetical rules_python implementation (e.g., py_binary.bzl)

# ... (other rule implementation details) ...

def _py_binary_impl(ctx):
    # ... (existing py_binary logic to determine files and their metadata) ...
    return [
        # DefaultInfo ...
        # ... other providers ...
        FSManifestInfo( .... ),
    ]

py_binary = rule(
    implementation = _py_binary_impl,
    # ... (other attributes) ...
    provides = [FSManifestInfo],
)
```

```python
# BUILD.bazel

load("@rules_python//python:defs.bzl", "py_binary")
load("@rules_img//img:defs.bzl", "image_layer")

# A sample Python binary
py_binary(
    name = "my_app",
    src = "main.py",
    # ...
)

# An image layer consuming the "what-goes-where" information from my_app
image_layer(
    name = "app_layer",
    base = "@ubuntu",
    # The new attribute to consume the "what-goes-where" provider
    bundles = [":my_app]",
    # Other image_layer specific attributes
    user = "appuser",
    working_dir = "/app",
)
```

```
# A ficticious rule that doesn't exist yet, but that can make use of the same
"what-goes-where" information
deploy_to_s3(
    name = "deploy_s3",
    bucket = "contorso_production",
    bundles = [":my_app"],
)
```

Contending API Approaches

We propose two primary API approaches for the "what-goes-where" provider:

1. `mtree` Format

The first approach leverages the `mtree` format, a well-defined standard for describing file hierarchies and their attributes. The provider should allow rules to produce multiple mtree files that can be used to produce separate filesystem layouts.

Advantages of `mtree`

- **Well-defined format:** `mtree` is a recognized standard understood by existing tools, offering clear definitions of file placement and support for metadata like permissions, owner, and flags.
- **Action-based introspection:** Being the result of an action, an `mtree` file producer can inspect file contents and make intelligent decisions not possible from Starlark, such as examining individual files within a `TreeArtifact`.

Disadvantages of `mtree`

- **Opaque to analysis phase:** An `mtree` file is produced by an action, meaning it can only be read by another action, not directly by a Starlark rule implementation during the analysis phase.
- **Path materialization:** `mtree` requires inputs to be referred to by their paths written to the `mtree` file. This could pose challenges for path mapping scenarios where input file paths might differ in various contexts. This is unlikely to be a concern in reality, since anything consuming the mtree file is unlikely to make use of pathmapping.
- **Limited metadata support:** While `mtree` handles common file metadata, it may not support extended attributes. For example, `go-mtree`'s format (refer to

https://github.com/vbatts/go-mtree?tab=readme-ov-file#format) required non-standard extensions to handle xattrs. How would we convey extra metadata not supported by the BSD standard and most tools?

2. Starlark Provider with Dictionary Mapping

You can find a work-in-progress implementation of this approach here:

- **malt3/bazel-what-goes-where-experiment: Overview containing an ecosystem of modules (with patches to existing modules) and examples**
- **malt3/runfilesgroupinfo.bzl: Provider for splitting the runfiles of a binary into smaller groups**
- **malt3/fsmanifestinfo.bzl: Provider for specifying placement and metadata of individual `File` objects**

The second approach proposes two new Starlark providers:

- RunfilesGroupInfo
- FSManifestInfo

Those providers solve the following problems:

RunfilesGroupInfo Provider

`RunfilesGroupInfo` is essentially the same as `OutputGroupInfo`, but with special semantics. If `RunfilesGroupInfo` is present next to `DefaultInfo` for a target, then the different groups in `RunfilesGroupInfo` can be used instead of the default runfiles (in `DefaultInfo.default_runfiles`). It is assumed that merging all the depsets contained in the runfiles group gives you the full set of default runfiles back, so this provider can be used to subdivide the runfiles into categories.
The following well-known groups are defined:

- SAME_PARTY_RUNFILES: Runfiles from the same package needed at runtime. This is typically the main binary and any data files it needs. This is also considered the "default" category if no other, more specific category applies.
- OTHER_PARTY_RUNFILES: Runfiles from third-party dependencies needed at runtime. These are typically shared libraries, interpreted code, or other resources that are not part of the same package as the target. This can be considered to be the default category for files provided by external repositories if no other, more specific category applies.
- FOUNDATIONAL_RUNFILES: Runfiles that are foundational to the application, e.g., interpreter or standard libraries. These can typically be shared across multiple applications. Sometimes they can be substituted by the runtime environment.
- DEBUG_RUNFILES: Runfiles needed for debugging the application. These are typically not needed for normal operation but can be useful for diagnosing issues.

This can include external ELF files, DWARF files, source maps, or other external debug symbols.
- DOCUMENTATION_RUNFILES: Runfiles needed for documentation. The application SHOULD still function without these files. Features that rely on these files MAY be disabled if they are not present, including help commands or man pages.

Consider the following usage example:

`foo_binary.bzl`:

```python
load("@runfilesgroupinfo.bzl", "RunfilesGroupInfo", "SAME_PARTY_RUNFILES",
"OTHER_PARTY_RUNFILES", "FOUNDATIONAL_RUNFILES", "DEBUG_RUNFILES")

def _foo_binary_impl(ctx):
    # ... skipping over the rest of the logic

    # The runfiles contain all groups merged into one
    runfiles = ctx.runfiles(
        files = [main_file, interpreter_executable],
        transitive_files = [
            same_party_transitive,
            third_party_transitive,
            standard_library_data,
            language_runtime_data,
            sourcemaps,
            external_dwarf_symbols,
        ],
    )

    # RunfilesGroupInfo keeps different groups separated
    runfiles_group_info = RunfilesGroupInfo(
        SAME_PARTY_RUNFILES = depset([main_file], transitive =
[same_party_transitive]),
        OTHER_PARTY_RUNFILES = depset(transitive = [third_party_transitive]),
        FOUNDATIONAL_RUNFILES = depset([interpreter_executable], transitive =
[standard_library_data, language_runtime_data]),
        DEBUG_RUNFILES = depset(transitive = [sourcemaps,
external_dwarf_symbols]),
    )

    # We return DefaultInfo and RunfilesGroupInfo.
    # This allows for the following use-cases:
```

```
    #  - consumers that cannot interpret RunfilesGroupInfo can simply ignore it
and just use DefaultInfo (and the runfiles inside of it)
    #  - consumers that are aware of RunfilesGroupInfo can make informed
decisions about splitting and filtering the different groups.
    #
    # Consumers that are aware of RunfilesGroupInfo can treat it as an optional
provider. If it is not present, they can still fall back to the full set of
runfiles from DefaultInfo.
    return [
        DefaultInfo(..., runfiles = runfiles),
        runfiles_group_info,
    ]

foo_binary = rule(
    implementation = _foo_binary_impl,
    # ... other arguments omitted
)
```

`custom_packaging_ruleset.bzl`:

```python
load("@runfilesgroupinfo.bzl", "RunfilesGroupInfo", "SAME_PARTY_RUNFILES",
"OTHER_PARTY_RUNFILES", "FOUNDATIONAL_RUNFILES", "DEBUG_RUNFILES")

def _custom_package_impl(ctx):
    default_info = ctx.attr.binary[DefaultInfo]
    if RunfilesGroupInfo in ctx.attr.binary:
        # perform custom logic like splitting the third party deps into a
separate layer
        # ... or maybe omit DEBUG_RUNFILES if compilation mode is not dbg
        runfiles_group_info = ctx.attr.binary[RunfilesGroupInfo]

        # use default_info.files together with some subset of
runfiles_group_info
        return ...

    # if RunfilesGroupInfo is missing, we can still process the full set of
runfiles as a fallback
    # do something with default_info.files and default_info.default_runfiles
    return ...
```

FSManifestInfo Provider

`FSManifestInfo` contains a dictionary that maps "path in image" to a Bazel `File` object (we could potentially also allow a depset of `File` or a runfiles object to be used as the value), along with file metadata (in JSON or another TBD format). This approach is similar to `rules_pkg`'s [`PackageFilesInfo`].

Take a look at the [draft implementation] for more information.

Advantages of Starlark Providers

- **Remote Execution-Friendly Approach to Layer Splitting:** Current implementations for splitting runfiles of binaries into different container image layer make use of "large" actions that are provided with all runfiles of a binary. The unused files are then reported via `unused_inputs_list`. This doesn't work well with remote execution, since unused inputs still need to be available to the remote executor. The remote cache also is not capable of "stripping" unused inputs. `RunfilesGroupInfo` provides a memory-efficient alternative that allows for efficient splitting of runfiles into separate actions, where each action only sees the runfiles it needs.
- **Pure and inspectable:** Similar to `rules_pkg`'s [`PackageFilesInfo`], this can be produced by a rule without running an action (it's pure) and can be inspected and merged by a downstream rule.
- **Direct file access:** This format already contains the actual `File` structs, eliminating the need to pass `mtree` and files separately.
- **No path materialization:** It doesn't materialize the paths of the inputs, which is beneficial for path mapping scenarios where input file paths might be dynamic. This benefit is mostly theoretical.

Disadvantages of Starlark Provider

- **Limited visibility into directories (TreeArtifacts):** The provider would consist of `File` structs that are mapped to paths in the deliverable with data known during the analysis phase. At this stage, tree artifacts (the output of `ctx.actions.declare_directory`), are just a `File` with `is_directory == True`. It's impossible to know what files are in a tree artifact during the analysis phase.
- **Metadata handling:** If using a Starlark dictionary for path-to-file mapping, we need to carefully consider how to handle and encode metadata in a clear and well-understood manner. A possible encoding would be a second dictionary containing a path-to-metadata mapping, where the metadata uses a JSON

encoding of the metadata, which requires standardization across Bazel rules to work effectively.

3. Annotating edges and categorizing files via aspects

A third approach proposes a technique where dependency edges are annotated with semantic information about the kind of dependency (e.g., the boundary between 1st-party and 3rd-party dependencies), which could be used by an aspect to split the files up into multiple components.

This approach is still in a very early draft stage and subject to change.

An edge could be annotated with well-known semantic tags, including at least the following:

- “embeds/runtime dependency” (included in deliverable) vs “Build against/compile dependency” (excluded from deliverable)
- “Same party dependency” (implicit by not being separated by an annotated boundary) vs. “other party dependency” (separated by an actual annotated boundary)
    - Dependencies belonging to the same “zone” could be placed in the same subdeliverable (e.g., a container image layer), allowing for a separation of 1st-party and 3rd-party dependencies.
- Other annotations (including custom ones that may be introduced by the delivery method)

Conclusion

Both `mtree` and the Starlark provider with a dictionary mapping offer distinct advantages and disadvantages. While `mtree` provides a standardized, action-based approach with existing tool support, its opacity during the analysis phase and potential limitations with extended attributes are concerns. The Starlark provider, on the other hand, offers a pure, inspectable, and flexible approach with direct file access, but requires careful design for metadata encoding.

Further discussion and detailed API design are needed to determine the most suitable approach for the "what-goes-where" provider, taking into account extensibility, performance, and ease of use for various language rulesets and packaging/deployment scenarios.