Pulumi

Pulumi

Oct 14, 2023

SYSTEM ARCHITECTURE

1 Pulumi Architecture Overview **3**
- 1.1 The Deployment Engine . . . 3
- 1.2 State Storage Backends . . . 3
- 1.3 Language SDKs . . . 3
- 1.4 Resource Providers . . . 3
- 1.5 Package Schemas and Code Generators . . . 3

2 Resource Registration **5**
- 2.1 The Resource Monitor . . . 5
- 2.2 The Step Generator . . . 7
- 2.3 The Step Executor . . . 9
- 2.4 Example Resource Registration Sequences . . . 9

3 Deployment Schema **11**
- 3.1 Pulumi Deployment States . . . 11
- 3.2 Pulumi Property Value . . . 16
- 3.3 Pulumi Resource State . . . 21

4 Pulumi Type System **27**
- 4.1 Primitive Types . . . 27
- 4.2 Object Types . . . 28
- 4.3 `Promise<T>` . . . 28
- 4.4 `Output<T>` . . . 28
- 4.5 `Input<T>` . . . 29
- 4.6 `inputShape(T)` . . . 29
- 4.7 `outputShape(T)` . . . 30
- 4.8 `plainShape(T)` . . . 31
- 4.9 `Output<T>` Combinators . . . 31

5 Importing Resources **33**
- 5.1 `import` resource option . . . 33
- 5.2 `pulumi import` . . . 34

6 Resource Provider Implementer's Guide **37**
- 6.1 Provider Programming Model . . . 37
- 6.2 Schema . . . 39
- 6.3 Provider Lifecycle . . . 40
- 6.4 Custom Resource Lifecycle . . . 42
- 6.5 Component Resource Lifecycle . . . 47
- 6.6 Provider Functions . . . 47
- 6.7 CLI Scenarios . . . 47

6.8 Appendix 48

7 Pulumi Package Metaschema **49**

7.1 Properties 49
7.2 Alias Definition 53
7.3 Array Type 54
7.4 Enum Type Definition 54
7.5 Enum Value Definition 55
7.6 Function Definition 56
7.7 Map Type 57
7.8 Named Type 57
7.9 Object Type Definition 58
7.10 Object Type Details 58
7.11 Primitive Type 59
7.12 Property Definition 59
7.13 Resource Definition 61
7.14 Token 63
7.15 Type Definition 63
7.16 Type Reference 64
7.17 Union Type 64

8 Building the Docs **67**

8.1 Notes on Style 67

Welcome to the Pulumi developer documentation! This documentation provides details on Pulumi internals, including but not limited to:

- How to build and test Pulumi
- Pulumi architectural details
- Specifications

Use the navigation bar to the left to browse the docs.

CHAPTER

ONE

PULUMI ARCHITECTURE OVERVIEW

Broadly speaking, Pulumi is composed of five components:

1. *The deployment engine*
2. *State storage backends*
3. *Language SDKs*
4. *Resource providers*
5. *Package schemas and code generators*

These components interact to provide the feature set exposed by the Pulumi CLI and SDKs, including desired-state deployments using standard programming languages, remote state storage and secret encryption, and the ability to bridge the gap between existing and Pulumi-managed infrastructure.

These components are composed like so:

In most cases, the language plugin, CLI, and resource providers will all live in separate processes, and each instance of a resource provider will live in its own process.

1.1 The Deployment Engine

1.2 State Storage Backends

1.3 Language SDKs

1.4 Resource Providers

1.5 Package Schemas and Code Generators

CHAPTER

TWO

RESOURCE REGISTRATION

A Pulumi program declares the desired states of its stack's resources by sending `RegisterResource` requests to the Pulumi engine. Each `RegisterResource` request contains the type, name, and parent (if any) of the resource, a reference to the provider instance that manages the resource (where an empty reference indicates that the resource uses the default provider instance for its package + version), the values of the resource's input properties, and any options that apply to the resource. The engine decides what step to take in order to drive a resource to its goal state by diffing the resource's current state as present in the statefile with its desired state. If there is no current state, the resource is created. Otherwise, the engine calls the resource's provider's `Diff` method to determine wither the resource is unchanged, updated, or replaced. Once the required action (or actions, in the case of a replacement) has been determined, the engine calls the resource's provider's `Create`, `Update`, or `Delete` methods to perform it. After the action completes, the engine returns the new state of the resource to the Pulumi program.

Although we typically treat the engine as a single unit, in the case of resource registrations it helps to break it down into a few of its internal components: the *resource monitor*, the *step generator*, and the *step executor*. Each of these components is a participant in the response to a `RegisterResourceRequest`.

2.1 The Resource Monitor

The *resource monitor* provider serves the `ResourceMonitor` gRPC interface, and provides a shim between language SDKs and the rest of the engine. There is a single resource monitor per deployment. As the engine's feature set has grown, the resource monitor has taken on responsibilities beyond its original use as a simple marshaling/unmarshaling layer. It is now responsible for handling default providers (providers for resource registrations that do not reference a provider instance) and for dispatching `RegisterResourceRequests` for multi-language components into appropriate `Construct` calls.

When the resource monitor receives a resource registration, it does the following:

1. Unmarshals data from the gRPC wire format to the engine's internal representation.
2. If the registration request does not name a provider instance, handles the resolution of the resource's default provider.
3. If the request is for a multi-language component, dispatches a `Construct` call to the component's provider and waits for the result.
4. If the request is not for a multi-langauge component, sends a `RegisterResourceEvent` to the *step generator* and waits for the result.
5. Marshals the result of the `Construct` call or `RegisterResourceEvent` from the engine's internal representation to the gRPC wire format and returns from the RPC call.

2.1.1 Default Providers

Default providers demand some amount of special attention. A *default provider* for a package and version is the provider instance that is used for resources at that package and version that do not otherwise reference a provider instance when they are registered. For example, consider the following program that creates an AWS S3 bucket:

```
import * as aws from "@pulumi/aws";

new aws.s3.Bucket("myBucket");
```

The constructor call will become a `RegisterResourceRequest` like:

```
RegisterResourceRequest{
        type: "aws:s3/bucket:Bucket",
        name: "myBucket",
        parent: "urn:pulumi:dev::project::pulumi:pulumi:Stack::project",
        custom: true,
        object: {},
        version: "4.16.0",
}
```

Because this request does not contain a value for the `provider` field, it will use the default provider for the `aws` package at version 4.16.0. The resource monitor ensures that only a single default provider instance exists for each particular package version, and only creates default provider instances if they are needed. Default provider instances are registered by synthesizing an appropriate `RegisterResourceEvent` with input properties sourced from the stack's configuration values for the provider's package. In the example above, the AWS default provider would be configured using any stack configuration values whose keys begin with `aws:` (e.g. `aws:region`).

If we change the program slightly to create and reference a provider instance, the default provider will no longer be used:

```
import * as aws from "@pulumi/aws";

const usWest2 = new aws.Provider("us-west-2", {region: "us-west-2"});

new aws.s3.Bucket("myBucket", {}, {provider: usWest2});
```

The constructor call will become a `RegisterResourceRequest` like:

```
RegisterResourceRequest{
        type: "aws:s3/bucket:Bucket",
        name: "myBucket",
        parent: "urn:pulumi:dev::project::pulumi:pulumi:Stack::project",
        custom: true,
        object: {},
        provider: "urn:pulumi:dev::vpc-2::pulumi:providers:aws::us-west-2::308b79ee-8249-
↪40fb-a203-de190cb8faa8",
        version: "4.16.0",
}
```

Note that this request *does* contain a value for the `provider` field.

2.2 The Step Generator

The *step generator* is responsible for processing `RegisterResourceEvent`s from the *resource monitor*. The generator implements the core logic that determines which actions to take in order to drive the actual state of a resource to its desired state as represented by the input properties in its `RegisterResourceEvent`. In order to simplify reasoning about the actual state of a stack's resources, the step generator processes `RegisterResourceEvent`s serially. It is important to note that this approach puts the step generator on a deployment's critical path, so any significant blocking in the step generator may slow down deployments accordingly. In the case of updates, step generator latency is generally insignificant compared to the time spent performing resource operations, but this is not the case for updates where most resources are unchanged or for previews, which spend very little time in resource providers in general.

When the step generator receives a `RegisterResourceEvent`, it does the following:

1. Generate a URN for the resource using the resource's type, name, and parent.
2. Look up the existing state for the resource, if any. If the event contains aliases for the resource, this includes checking for existing state under those aliases. It is an error if a resource's aliases match multiple existing states.
3. Pre-process input properties for ignored changes by setting any properties mentioned in the event's ignore changes list to their old value (if any)
4. If the event indicates that the resource should be imported, issue an `ImportStep` to the *step executor* and return.
5. Call the resource's provider's `Check` method with the event's input properties and the resource's existing inputs, if any. The existing inputs may be used by the provider to repopulate default values for input properties that are automatically generated when the resource is created but should not be changed with subsequent updates (e.g. automatically generated names). `Check` returns a pre-processed bag of input values to be used with later calls to `Diff`, `Create`, and `Update`.
6. Invoke any analyzers for the stack to perform additional validation of the resource's input properties.
7. If the resource has no existing state, it is being created. Issue a `CreateStep` to the *step executor* and return.
8. Diff the resource in order to determine whether it must be updated, replaced, delete-before-replaced, or has no changes. Diffing is covered in detail later on, but typically consists of calling the reosource's provider's `Diff` method with the checked inputs from step 5.
9. If the resource has no changes, issue a `SameStep` to the *step executor* and return.
10. If the resource is not being replaced, issue an `UpdateStep` to the *step executor* and return.
11. If the resource is being replaced, call the resource's provider's `Check` method again, but with no existing inputs. This call ensures that the input properties used to create the replacement resource do not reuse generated defaults from the existing resource.
12. If the replacement resource is being created before the original is deleted (a normal replacement), issue a `CreateStep` and a `DeleteStep` to the *step executor* and return.
13. At this point, the resource must be deleted before its replacement is created (this is the "delete-before-replace" case). Calculate the set of dependent resources that must be deleted prior to deleting the resource being replaced. The details of this calculation are covered in a later section. Once the set of deletions has been calculated, issue a sequence of `DeleteStep`s followed by a single `CreateStep` to the *step executor*.

Note that all steps that are issued to the step generator are fire-and-forget. Once steps have been issues, the step generator moves on to process the next `RegisterResourceEvent`. It is the responsibility of the *step executor* to communicate the results of each step back to the *resource monitor*.

Once the Pulumi program has exited, the step generator determines which existing resources must be deleted by taking the difference between the set of registered resources and the set of existing resources. These resources are scheduled for deletion by first sorting the list of resources to delete using the topological order of their reverse-dependency grapth, then decomposing the list into a list of lists where each list can be executed in parallel but a previous list must be executed to completion before advancing to the next list.

In lieu of tracking per-step dependencies and orienting the *step executor* around these dependencies, this approach provides a conservative approximation of which deletions can safely occur in parallel. The insight here is that the resource dependency graph is a partially-ordered set and all partially-ordered sets can be easily decomposed into antichains–subsets of the set that are all not comparable to one another (in this definition, “not comparable” means “do not depend on one another”).

The algorithm for decomposing a poset into antichains is:

1. While there exist elements in the poset, a. There must exist at least one “maximal” element of the poset. Let `E_max` be those elements. b. Remove all elements E_max from the poset. `E_max` is an antichain. c. Goto 1.

Translated to a resource dependency graph:

1. While the set of condemned resources is not empty: a. Remove all resources with no outgoing edges from the graph and add them to the current antichain. b. Goto 1.

The resulting list of antichains is a list of list of delete steps that can be safely executed in parallel. Since deletes must be processed in reverse order (so that resources are not deleted prior to their dependents), the step generator reverses the list and then issues each sublist to the *step executor*.

2.2.1 Resource Diffing

Although resource diffing is simple in most cases, there are several possibilities that the step generator must consider as part of performing a diff. The algorithm for diffing a resource is outlined here.

1. If the resource has been marked for replacement out of band (e.g. by the use of the `--target-replace` command-line option of the Pulumi CLI), the resource must be replaced.
2. If the resource’s provider has changed, the resource must be replaced. Default providers are allowed to change without requiring replacement if and only if the provider’s configuration allows the new default provider to continue to manage existing resources (this is intended to allow default providers to be upgraded without requiring that all the resources they manage are replaced).
3. If the engine is configured to use pre-1.0-style diffs, compare the resource’s old and new inputs. If the old and new inputs differ, the resource must be updated.
4. Otherwise, call the resource’s provider’s `Diff` method with the resource’s new inputs, old state, and ignore changes set to determine whether the resource has changed, and if so, if it must be replaced.

Once the diff has been calculated, the step generator applies any replace-on-change options specified by the resource. These options force a resource to require that it is replaced if any of a particular set of properties has changed.

2.2.2 Dependent Replacements

When a resource must be deleted before it is replaced–whether this is required by the resource’s provider or is forced using the `deleteBeforeReplace` resource option–it may be necessary to first delete dependent resources. The step generator does this by taking the complete set of transitive dependents on the resource under consideration and removing any resources that would not be replaced by changes to their dependencies. It determines whether or not a resource must be replaced by substituting unknowns for any input properties that may change due to deletion of the resources their value depends on and calling the resource’s provider’s `Diff` method.

This is perhaps clearer when described by example. Consider the following dependency graph:

In this graph, all of B, C, D, E, and F transitively depend on A. It may be the case, however, that changes to the specific properties of any of those resources R that would occur if a resource on the path to A were deleted and recreated may not cause R to be replaced. For example, the edge from B to A may be a simple `dependsOn` edge such that a change to B does not actually influence any of B's input properties. More commonly, the edge from B to A may be due to a property from A being used as the input to a property of B that does not require B to be replaced upon a change. In these cases, neither B nor D would need to be deleted before A could be deleted.

2.3 The Step Executor

The *step executor* is responsible for executing sequences of steps (called "chains") that perform the resource actions for a deployment. These chains are issued by the *step generator*, and most often consist of a single step. While the steps the make up a chain must be performed serially, chains may be executed in parallel. The step executor uses a (potentially infinite) pool of workers to execute steps. Once a step completes, the step executor communicates its results to the *resource monitor* if necessary. If a step fails, the step executor notes the failure and cancels the deployment. Once the Pulumi program has exited and the *step generator* has issued all required deletions, the step executor waits for all outstanding steps to complete and then returns.

2.4 Example Resource Registration Sequences

2.4.1 Custom Resources

Each of the diagrams below demonstrates a sequence of events that occur when a custom resource is registered. Examples are given for each possible action: create, update, replace, delete-before-replace, import, and no change.

Create

Update

Replace

Delete-before-replace

Import

No change

2.4.2 Multi-language Components

The diagram below illustrates the sequence of events that occurs when a multi-language component is registered. The registration of the component's children is elided.

CHAPTER

THREE

DEPLOYMENT SCHEMA

3.1 Pulumi Deployment States

A schema for Pulumi deployment states.

`object`

One of:

3.1.1 Properties

`deployment` (*required*)

The deployment object.

`object`

`version` (*required*)

The deployment version.

`integer`

3.1.2 Deployment Manifest

Captures meta-information about a deployment, such as versions of binaries, etc.

`object`

Properties

`magic` (*required*)

A magic number used to validate the manifest's integrity.

`string`

`plugins`

Information about the plugins used by the deployment.

`array`

Items: *Plugin Info*

`time` (*required*)

The deployment's start time.

`string`

Format: `date-time`

`version` (*required*)

The version of the Pulumi engine that produced the deployment.

`string`

3.1.3 Plugin Info

Information about a plugin.

`object`

Properties

`name` (*required*)

The plugin's name.

`string`

`path` (*required*)

The path of the plugin's binary.

`string`

`type` (*required*)

The plugin's type.

Enum: `"analyzer"` | `"language"` | `"resource"`

`version` (*required*)

The plugin's version.

`string`

3.1.4 Resource Operation V2

Version 2 of a resource operation state

`object`

Properties

`resource` (*required*)

The state of the affected resource as of the start of this operation.

Resource V3

`type` (*required*)

A string representation of the operation.

Enum: `"creating"` | `"updating"` | `"deleting"` | `"reading"`

3.1.5 Secrets Provider

Configuration information for a secrets provider.

`object`

Properties

`state`

The secrets provider's state, if any.

`type` (*required*)

The secrets provider's type.

`string`

3.1.6 Unknown Version

Catchall for unknown deployment versions.

`object`

Properties

`deployment`

The deployment object.

`object`

`version`

The deployment version.

3.1.7 Version 3

The third version of the deployment state.

`object`

Properties

`deployment` (*required*)

The deployment state.

`object`

Properties

####### `manifest` (*required*)

Metadata about the deployment.

Deployment Manifest

####### `pending_operations`

Any operations that were pending at the time the deployment finished.

`array`

Items: *Resource Operation V2*

####### `resources`

All resources that are part of the stack.

`array`

Items: Resource V3

####### `secrets_providers`

Configuration for this stack's secrets provider.

Secrets Provider

`version` (***required***)

The deployment version. Must be `3`.

Constant: `3`

3.2 Pulumi Property Value

A schema for Pulumi Property values.

One of:

3.2.1 Archive property values

`object`

One of:

Properties

`4dabf18193072939515e22adb298388d` (***required***)

Archive signature

Constant: `"0def7320c3a5731c473e5ecbe6d01bc7"`

`hash`

The SHA256 hash of the archive's contents.

`string`

3.2.2 Array property values

`array`

Items: *Pulumi Property Value*

3.2.3 Asset property values

`object`

One of:

Properties

`4dabf18193072939515e22adb298388d` (*required*)

Asset signature

Constant: `"c44067f5952c0a294b673a41bacd8c17"`

`hash`

The SHA256 hash of the asset's contents.

`string`

3.2.4 Decrypted Secret

`object`

Properties

`plaintext` (*required*)

The decrypted, JSON-serialized property value

`string`

3.2.5 Encrypted Secret

`object`

Properties

`ciphertext` (*required*)

The encrypted, JSON-serialized property value

`string`

3.2.6 Hash-only Archive

3.2.7 Hash-only Asset

3.2.8 Literal Archive

Properties

`assets` (*required*)

The literal contents of the archive.

`object`

Additional properties: *https://github.com/pulumi/pulumi/blob/master/sdk/go/common/apitype/property-values.json#/oneOf/5/oneOf/1/properties/assets/additionalProperties*

3.2.9 Literal Asset

Properties

`text` (*required*)

The literal contents of the asset.

`string`

3.2.10 Local File Archive

Properties

`path` (***required***)

The path to a local file that contains the archive's contents.

`string`

3.2.11 Local File Asset

Properties

`path` (***required***)

The path to a local file that contains the asset's contents.

`string`

3.2.12 Object property values

`object`

Additional properties: *Pulumi Property Value*

3.2.13 Primitive property values

`null` | `boolean` | `number` | `string`

3.2.14 Pulumi Property Value

A schema for Pulumi Property values.

One of:

3.2.15 Resource reference property values

`object`

Properties

`4dabf18193072939515e22adb298388d` (*required*)

Resource reference signature

Constant: `"5cf8f73096256a8f31e491e813e4eb8e"`

`id`

The ID of the referenced resource.

`string`

`packageVersion`

The package version of the referenced resource.

`string`

`urn` (*required*)

The URN of the referenced resource.

`string`

3.2.16 Secret Property Values

`object`

One of:

Properties

`4dabf18193072939515e22adb298388d` (*required*)

Secret signature

Constant: `"1b47061264138c4ac30d75fd1eb44270"`

3.2.17 URI File Archive

Properties

`uri` (***required***)

The URI of a file that contains the archive's contents.

`string`

Format: `uri`

3.2.18 URI File Asset

Properties

`uri` (***required***)

The URI of a file that contains the asset's contents.

`string`

Format: `uri`

3.2.19 Unknown property values

Constant: `"04da6b54-80e4-46f7-96ec-b56ff0331ba9"`

3.2.20 `https://github.com/pulumi/pulumi/blob/master/sdk/go/common/apitype/property-values.json#/oneOf/5/oneOf/1/properties/assets/additionalProperties`

One of:

3.3 Pulumi Resource State

Schemas for Pulumi resource states.

One of:

3.3.1 Resource V3

Version 3 of a Pulumi resource state.

`object`

Properties

`additionalSecretOutputs`

A list of outputs that were explicitly marked as secret when the resource was created.

`array`

Items: `string`

`aliases`

A list of previous URNs that this resource may have had in previous deployments

`array`

Items: *Unique Resource Name (URN)*

`custom`

True when the resource is managed by a plugin.

`boolean`

`customTimeouts`

A configuration block that can be used to control timeouts of CRUD operations

`object`

`delete`

True when the resource should be deleted during the next update.

`boolean`

---

dependencies

The dependency edges to other resources that this depends on.

`array`

Items: *Unique Resource Name (URN)*

---

external

True when the lifecycle of this resource is not managed by Pulumi.

`boolean`

---

id

The provider-assigned resource ID, if any, for custom resources.

`string`

---

importID

The import input used for imported resources.

`string`

---

initErrors

The set of errors encountered in the process of initializing resource (i.e. during create or update).

`array`

Items: `string`

---

inputs

The input properties supplied to the provider.

`object`

Additional properties: Pulumi Property Value

`outputs`

The output properties returned by the provider after provisioning.

`object`

Additional properties: Pulumi Property Value

`parent`

An optional parent URN if this resource is a child of it.

Unique Resource Name (URN)

`pendingReplacement`

Tracks delete-before-replace resources that have been deleted but not yet recreated.

`boolean`

`propertyDependencies`

A map from each input property name to the set of resources that property depends on.

`object`

Additional properties: *https://github.com/pulumi/pulumi/blob/master/sdk/go/common/apitype/resources.json#/$defs/resourceV3/properties/propertyDependencies/additionalProperties*

`protect`

True when this resource is "protected" and may not be deleted.

`boolean`

`provider`

A reference to the provider that is associated with this resource.

`string`

`type`

The resource's full type token.

`string`

`urn` (***required***)

The resource's unique name.

Unique Resource Name (URN)

3.3.2 Unique Resource Name (URN)

The unique name for a resource in a Pulumi stack.

`string`

3.3.3 `https://github.com/pulumi/pulumi/blob/master/sdk/go/common/apitype/resources.json#/$defs/resourceV3/properties/propertyDependencies/additionalProperties`

`array`

Items: *Unique Resource Name (URN)*

CHAPTER

FOUR

PULUMI TYPE SYSTEM

In its role as a broker of information between various actors–e.g. language SDKs, resource providers, multi-language components, and statefiles–and in its role as a programming model, it is important that Pulumi deals in values with well-defined semantics. The *Pulumi type system* specifies these semantics. It is the responsibility of each language SDK and interchange format to ensure that these semantics are faithfully implemented, ideally in as idiomatic a fashion as possible.

Note that this document describes the abstract type system rather than describing its implementations. As long as implementations faithfully implement the semantics described by this document, they may choose to provide simpler APIs/shorthands/etc. for the various types and combinators. For example, the SDK for a language that natively supports operations on *`Output<T>`* values may not expose *`Output<T>`* types and combinators to the user at all.

4.1 Primitive Types

The core primitives of the Pulumi type system form a superset of the JSON type system, and supports the following types:

- `Null`, which represents the lack of a value
- `Bool`, which represents a boolean value
- `Number`, which represents an IEEE-754 double-precision number
- `String`, which represents a sequence of UTF-8 encoded unicode code points
- *`Asset`*, which represents a blob
- *`Archive`*, which represents a map from strings to `Assets` or `Archives`
- *`ResourceReference`*, which represents a reference to a resource
- `Tuple<T0, T1, ... TN>`, which represents a tuple of heterogenously-typed values. Note that this type mainly exists for the purpose of writing the signature for *`all`*.
- `Array<T>`, which represents a numbered sequence of values of a particular type
- `Map<T>`, which represents an unordered mapping from strings to values of a particular type
- `Union<T0, T1 ... TN>`, which represents a value of one of a fixed set of types
- `Enum<T, V0 ... VN>`, which represents one of a fixed set of values of a particular type

4.1.1 Assets and Archives

An `Asset` or `Archive` may contain either literal data or a reference to a local file located via its path or a local or remote file located via its URL.

In the case of `Asset`s, the literal data is a textual string, and the referenced file is an opaque blob.

In the case of `Archive`s, the literal data is a map from strings to `Asset`s or `Archive`s, and the referenced file is a TAR archive, gzipped TAR archive, or ZIP archive.

Each `Asset` or `Archive` also carries the SHA-256 hash of its contents. This hash can be used to uniquely identify the asset (e.g. for locally caching `Asset` or `Archive` contents).

4.1.2 Resource References

A `ResourceReference` represents a reference to a resource. Although all that is necessary to uniquely identify a resource within the context of a stack is its URN, a `ResourceReference` also carries the resource's ID (if it is not a component) and the version of the provider that manages the resource. If the contents of the referenced resource must be inspected, the reference must be resolved by invoking the `getResource` function of the engine's builtin provider. Note that this is only possible if there is a connection to the engine's resource monitor, e.g. within the scope of a call to `Construct`. This implies that resource references may not be resolved within calls to other provider methods. Therefore, configuration values, custom resources and provider functions should not rely on the ability to resolve resource references, and should instead treat resource references as either their ID (if present) or URN. If the ID is present and empty, it should be treated as an *`Unknown`*.

4.2 Object Types

Object types are defined as mapping from property names to property types. Duplicate property names are not allowed, and each property name maps to a single type.

4.3 `Promise<T>`

A value of type `Promise<T>` represents the result of an asynchronous computation. Note that although computations may fail, failures cannot be handled at runtime, and cause a hard stop when attempting to access a `Promise<T>`'s concrete value.

4.4 `Output<T>`

Perhaps the most important type in the Pulumi type system is `Output<T>`. A value of type `Output<T>` represents a node in a Pulumi program graph, and behaves like a `Promise<T>` that carries additional metadata that describes the resources on which the value depends, whether the value is known or unknown, and whether or not the value is secret.

4.4.1 Dependencies

If an `Output<T>` value is the result of a resource operation–e.g. if it is an output property of some resource–it is said to *depend* on that resource.

If a value of type `Output<T>` depends on a resource `R`, the result of any computation that depends on its concrete value also depends on `R`.

4.4.2 Unknowns

An `Output<T>` may be unknown if it depends on the result of a resource operation that will not be run because it is part of a `pulumi preview`. Previews typically produce unknowns for properties with values that cannot be determined until the resource is actually created or updated.

If a value of type `Output<T>` is unknown, any computation that depends on its concrete value must not run, and must instead produce an unknown `Output<T>`.

4.4.3 Secrets

An `Output<T>` may be marked as secret if its concrete value contains sensitive information.

If a value of type `Output<T>` is secret, the result of any computation that depends on its concrete value must also be secret.

4.5 `Input<T>`

The partner of `Output<T>` is `Input<T>`, which is defined as `Union<T, Output<T>>`. In simpler terms, a location of type `Input<T>` may accept either a plain old `T` value or an `Output<T>` value.

4.6 `inputShape(T)`

Although `Input<T>` gives us the ability to deal in both `T` and `Output<T>` values, it is often the case that we want to construct *composite* values out of multiple `Input<T>`s. For example, consider `Input<Array<string>>`: a value of this type accepts either a `Array<string>` or an `Output<Array<string>>`, but does not accept a value of type `Array<Output<string>>`. In order to accept all three of these types, we need the type `Input<Array<Input<string>>>>`. The `inputShape` type function defines an algorithm for producing these sorts of types.

```
fn inputShape(T) {
        match T {
                _ => Input<T>,
                Tuple<...U> => Input<Tuple<map(...U, u => inputShape(u))>>,
                Array<U> => Input<Array<inputShape(U)>>,
                Map<U> => Input<Map<inputShape(U)>>,
                Union<...U> => Union<map(...U, u => inputShape(u))>,
                Promise<U> => Input<U>,
                Output<U> => Input<U>,
                Object<...P> => Input<Object<map(...P, (name, u) => (name,␣
↪inputShape(u))>>
```

(continues on next page)

(continued from previous page)

```
        }
}
```

If we expand `Input<T>` into its underlying type, `Union<T, Output<T>>`, the types may be clearer:

```
fn inputShape(T) {
        match T {
                _ => Union<T, Output<T>>,
                Tuple<...U> => Union<Tuple<map(...U, u => inputShape(u))>, Output<Tuple
↪<map(...U, u => inputShape(u))>>>,
                Array<U> => Union<Array<inputShape(U)>, Output<Array<inputShape(U)>>>,
                Map<U> => Union<Map<inputShape(U)>, Output<Map<inputShape(U)>>>,
                Union<...U> => Union<map(...U, u => inputShape(u))>,
                Promise<U> => Union<U, Output<U>>,
                Output<U> => Union<U, Output<U>>,
                Object<...P> => Union<Object<map(...P, (name, u) => (name,␣
↪inputShape(u)))>, Output<Object<map(...P, (name, u) => (name, inputShape(u))>>>
        }
}
```

Resource input properties often use input-shaped types.

4.7 `outputShape(T)`

Because the `Output<T>` metadata (*dependencies*, *unknowns*, and *secrets*) only applies to a single value, it is necessary to represent composite metadata using nested `Output<T>` types. Consider a variant of the `Array<string>` example from *`inputShape(T)`*: in order to produce an array where each element may be an output, we need to use the type `Output<Array<Output<string>>>`.

The `outputShape` type function defines an algorithm for producing these sorts of values.

```
fn outputShape(T) {
        match T {
                _ => Output<T>,
                Tuple<...U> => Output<Tuple<map(...U, u => outputShape(u))>>,
                Array<U> => Output<Array<outputShape(U)>>,
                Map<U> => Output<Map<outputShape(U)>>,
                Union<...U> => Union<map(...U, u => outputShape(u))>,
                Promise<U> => Output<U>,
                Output<U> => Output<U>,
                Object<...P> => Output<Object<map(...P, (name, u) => (name,␣
↪outputShape(u))>>
        }
}
```

Resource output properties often use output-shaped types.

Projecting output-shaped is a bit unwieldy, as values of these types often require a great deal of unwrapping as nesting depth increases.

Instead of projecting these types in their fully-elaborated form, the various language SDKs tend to opt to project them as a simple `Output<T>` while using an internal representation for the concrete value that includes distinguished unknown values. This approach lets the SDKs to allow e.g. lifted property and element access into partially-known composite values. For example, the Node SDK will allow the user to access an element of an `Output<[]string>` via a proxied index operator even if some elements of the array are unknown, though it will not allow the user to access the entire value via `apply`.

4.8 `plainShape(T)`

The final type function, `plainShape(T)`, replaces `Output<T>` types with their type argument:

```
fn plainShape(T) {
        match T {
                _ => T,
                Tuple<...U> => Tuple<map(...U, u => plainShape(u))>,
                Array<U> => Array<plainShape(U)>,
                Map<U> => Map<plainShape(U)>,
                Union<...U> => Union<map(...U, u => plainShape(u)),
                Promise<U> => U,
                Output<U> => U,
                Object<...P> => Object<map(...P, (name, u) => (name, plainShape(u))>,
        }
}
```

This function is primarily useful for describing the signature of the *`all`* combinator.

4.9 `Output<T>` Combinators

The rules described for working with `Output<T>` metadata–*dependencies*, *unknowns*, and *secrets*–require special bookkeeping on the part of the consumer. There are three primitive combinators that aid in this bookkeeping: *`apply`*, *`all`*, and *`unwrap`*.

4.9.1 `apply<T, U>(v: Output<T>, f: (T) => U): Output<U>`

The `apply` API allows its caller to access the concrete value of an `Output<T>` within the context of a caller-supplied callback.

`apply` trivially obeys the `Output<T>` rules for *dependencies*, *unknowns*, and *secrets*:

- the dependencies of the `Output<T>` argument are propagated to the result
- if the `Output<T>` argument is unknown, the callback is not run and the result is unknown
- if the `Output<T>` argument is secret, the result is secret

Note that the argument for `U` may itself be an `Output<V>`, in which case the return type of `apply` will be `Output<Output<V>>`. The result can be unwrapped using the *`unwrap`* combinator. A language SDK may opt to automatically unwrap such values if its type system is flexible enough to express the unwrapping.

This API is morally equivalent to Javascript's `Promise.then` API, but with `Output<>`s in the place of `Promise<>`s:

```
class Output<T> {
        public apply<U>(func: (t: T) => U): Output<U> {
                ...
        }
}
```

4.9.2 `unwrap<T>(v: Output<Output<T>>): Output<T>`

The `unwrap` API transforms an `Output<Output<T>>` into an `Output<T>` according to the `Output<T>` rules for *dependencies*, *unknowns*, and *secrets*:

- the result's dependencies are the union of the outer and inner `Output<>`s' dependencies
- if either the outer or inner `Output<>` is unknown, the result is unknowns
- if either the outer or inner `Output<>` is secret, the result is secret

If its type system is flexible enough, a language SDK may choose to omit a public-facing `unwrap` API in favor of automatically unwrapping nested `Output<>`s.

4.9.3 `all<T0 ... TN>(t0: Output<T0>, ... tn: Output<TN>): Output<plainShape(Tuple<T0 ... TN>)>`

The `all` API combines multiple heterogenous outputs into a single unwrapped tuple output. The metadata from the arguments is combined as per the `Output<T>` rules for *dependencies*, *unknowns*, and *secrets*:

- the result of `all` depends on the union of the dependencies of its `Output<>` arguments
- if any of the `Output<>` arguments is unknown, the result is unknown
- if any of the `Output<>` arguments is secret, the result is secret

For example, here is a simplified version of the signature for the Typescript implementation of `all`:

```
export function all<T1, T2>(values: [Output<T1>, Output<T2>]): Output<[Unwrap<T1>, Unwrap
↪<T2>]>;
```

As in *`apply`*, nested outputs must be unwrapped prior to use, though SDKs may choose to automatically unwrap if their type system can accommodate the typing.

A variant of `all` for `Objects` is also possible:

`all<Object<...P>>(v: Object<...P>): Output<plainShape(Object<...P>)>`

This variant treats the object as a tuple of key/value pairs.

CHAPTER

FIVE

IMPORTING RESOURCES

There are a variety of scenarios that require the ability for users to import existing resources for management by Pulumi. For example:

- Migrating from manually-managed resources to IaC
- Migrating from other IaC platforms to Pulumi
- Migrating resources between Pulumi stacks

At a minimum, importing a resource involves adding the resource's state to the destination stack's statefile. Once the resource has been added to the stack, the Pulumi CLI is able to manage the resource like any other. In order to do anything besides delete the resource, however, the user must also add a definition for the resource to their Pulumi program.

Both of the import approaches used by Pulumi aim to prevent the accidental modification or deletion of a resource being imported. Though the user experiences of these approaches are quite different, they share a common principle: at the point at which a resource is successfully imported, the stack's Pulumi program must contain a definition for the resource that accurately describes its current state (i.e. there are no differences between the state described in the program and the actual state of the imported resource).

5.1 `import` resource option

The oldest method supported of importing resources into a stack is the `import` resource option. When set, this option specifies the ID of an existing resource to import into the stack. The exact behavior of this option depends on the current state of the resource within the destination stack:

1. If the resource does not exist, it is imported
2. If the resource exists and has the same `ID` or `ImportID`, the resource is treated like any other resource
3. Otherwise, the current resource is deleted and replaced by importing the resource with the specified ID

The trickiest of these three situations is (2). This state transition is intended to allow users to import a resource and then continue to make changes to their program without requiring that they remove the resource option. For example, this allows a user to import a resource in one `pulumi up`, then successfully run another `pulumi up` without removing the `import` option from their program and without attempting to import the resource a second time.

As mentioned in *the introduction*, the `import` resource option requires that the desired state described by Pulumi program for a resource being imported matches the actual state of the resource as returned by the provider. More precisely, given a resource `R` of type `T` with import ID `X` and the resource inputs present in the Pulumi program `I`, the engine performs the following sequence of operations:

1. Fetch the current inputs `I` and state `S` for the resource of type `T` with ID `X` from its provider by calling the provider's *`Read` method*. If the provider does not return a value for `I`, the provider does not support importing resources and the import fails.

2. Process the `ignoreChanges` resource option by copying the value for any ignored input property from `I` to `I`.
3. Validate the resource's inputs and apply any programmatic defaults by passing `I` and `I` to the provider's *`Check` method*. Let `I` be the checked inputs; these inputs form the resource's desired state.
4. Check for differences between `I` and `S` by calling the provider's *`Diff` method*. If the provider reports any differences, the import either succeeds with a warning (in the case of a preview) or fails with an error (in the case of an update).

If all of these steps succeed, the user is left with a definition for `R` in their Pulumi program and the statefile of the updated stack that do not differ.

5.1.1 Technical Note

Although the "no diffs" requirement is intended to prevent surprise, it also accommodates a technical limitation of the Pulumi engine. In order to actually perform the diff–an operation that is required whether or not the user is permitted to describe a desired state for the imported resource that differs from its actual state–the engine must fetch the resource's current imports and state from its provider. In order for this state to affect the steps the engine issues for the resources, the state would need to be fetched during or prior to the point at which the resource's registration reaches the *step generator*. In the former case, this would cause the engine to spend an unacceptable amount of time in the step generator, as it processes resource registrations serially. In the latter case, the user experience would likely be negatively affected by a lack of output from the Pulumi CLI, which only displays the status of *steps*. In order to address these issues, the operations described above happen in a dedicated `ImportStep` that is run by the *step executor*.

5.2 `pulumi import`

The second, newer method of importing resources into a stack is the `pulumi import` command. This command accepts a list of import specs to import, imports the resources into the destination stack, and generates definitions for the resources in the language used by the stack's Pulumi program. Each import spec is at least a type token, name, and ID, but may also specify a parent URN, provider reference, and package version.

During a `pulumi import`, given a resource `R` of type `T` with import ID X and an empty set of input properties `I`, the engine performs the following sequence of operations:

1. Fetch the current inputs `I` and state `S` for the resource of type `T` with ID `X` from its provider by calling the provider's *`Read` method*. If the provider does not return a value for `I`, the provider does not support importing resources and the import fails.
2. Fetch the schema for resources of type `T` from the provider. If the provider is not schematized or if `T` has no schema, the import fails.
3. Copy the value of each required input property defined in the schema for `T` from `I` to `I`.
4. Validate the resource's inputs and apply any programmatic defaults by passing `I` and `I` to the provider's *`Check` method*. Let `I` be the checked inputs; these inputs form the resource's desired state.
5. Check for differences between `I` and `S` by calling the provider's *`Diff` method*. If the provider reports any differences, the values of the differing properties are copied from `S` to `I`. This is intended to produce the smallest valid set of inputs necessary to avoid diffs. This does not use a fixed-point algorithm because there is no guarantee that the values copied from `S` are in fact valid (state and inputs with the same property paths may have different types and validation rules) and there is no guarantee that such an algorithm would terminate (TF bridge providers have had bugs that cause persistent diffs, which can only be worked around with `ignoreChanges`).

If all of these steps succeed, the user is left with a definition for `R` in the statefile of the updated stack that do not differ. The Pulumi CLI then passes the inputs `I` stored in the statefile to the import code generator. The import code generator converts the values present in `I` into an equivalent PCL representation of `R`'s desired state, then passes the PCL to a language-specific code generator to emit a representation of `R`'s desired state in the language used by the destination stack's Pulumi program. The user can then copy the generated definition into their Pulumi program.

Graphically, the import process looks something like this:

5.2.1 Challenges

The primary challenge in generating appropriate code for `pulumi import` lies in determining exactly what the input values for a particular resource should be. In many providers, it is not necessarily possible to accurately recover a resource's inputs from its state. This observation led to the diff-oriented approach described above, where the importer begins with an extremely minimal set of inputs and attempts to derive the actual inputs from the results of a call to the provider's *`Diff` method*. Unfortunately, the results are not always satisfactory, and the relatively small set of inputs present in the generated code can make it difficult for users to determine what inputs they *actually* need to pass to the resource to describe its current state.

A few other approaches might be:

- Emit no properties at all; just appropriate constructor calls. This will almost always emit code that does not compile or run, as nearly every resource has at least one required property.
- Copy the value for every input property present in a resource's schema from its state. This risks emitting code that does not compile due to differences in types between inputs and outputs, and also risks emitting code that does not work at runtime due to conflicts between mutually-exclusive properties (these are common for TF-based resources, for example).

It is likely that some mix of approaches is necessary in order to arrive at a satisfactory solution, as none of the above solutions seems universally "correct".

CHAPTER

SIX

RESOURCE PROVIDER IMPLEMENTER'S GUIDE

6.1 Provider Programming Model

6.1.1 Resources

The core functionality of a resource provider is the management of custom resources and construction of component resources within the scope of a Pulumi stack. Custom resources have a well-defined lifecycle built around the differences between their acutal state and the desired state described by their inputs and implemented using create, read, update, and delete (CRUD) operations defined by the provider. Component resources have no associated lifecycle, and are constructed by registering child custom or component resources with the Pulumi engine.

URNs

Each resource registered with the Pulumi engine is logically identified by its uniform resource name (URN). A resource's URN is derived from the its type, parent type, and user-supplied name. Within the scope of a resource-related provider method (`Check`, `Diff`, `Create`, `Read`, `Update`, `Delete`, and `Construct`), the type of the resource can be extracted from the provided URN. The structure of a URN is defined by the grammar below.

```
urn = "urn:pulumi:" stack "::" project "::" qualified type name "::" name ;

stack   = string ;
project = string ;
name    = string ;
string  = (* any sequence of unicode code points that does not contain "::" *) ;

qualified type name = [ parent type "$" ] type ;
parent type         = type ;

type       = package ":" [ module ":" ] type name ;
package    = identifier ;
module     = identifier ;
type name  = identifier ;
identifier = unicode letter { unicode letter | unicode digit | "_" } ;
```

Custom Resources

In addition to its URN, each custom resource has an associated ID. This ID is opaque to the Pulumi engine, and is only meaningful to the provider as a means to identify a physical resource. The ID must be a string. The empty ID indicates that a resource's ID is not known because it has not yet been created. Critically, a custom resource has a *well-defined lifecycle* within the scope of a Pulumi stack.

Component Resources

A component resource is a logical conatiner for other resources. Besides its URN, a component resource has a set of inputs, a set of outputs, and a tree of children. Its only lifecycle semantics are those of its children; its inputs and outputs are not related in the same way a *custom resource's* inputs and state are related. The engine can call a resource provider's `Construct` method to request that the provider create a component resource of a particular type.

6.1.2 Functions

A provider function is a function implemented by a provider, and has access to any of the provider's state. Each function has a unique token, optionally accepts an input object, and optionally produces an output object. The data passed to and returned from a function must not be *unknown* or *secret*, and must not *refer to resources*. Note that an exception to these rules is made for component resource methods, which may accept values of any type, and are provided with a connection to the Pulumi engine.

6.1.3 Data Exchange Types

The values exchanged between Pulumi resource providers and the Pulumi engine are a superset of the values expressible in JSON.

Pulumi supports the following data types:

- `Null`, which represents the lack of a value
- `Bool`, which represents a boolean value
- `Number`, which represents an IEEE-754 double-precision number
- `String`, which represents a sequence of UTF-8 encoded unicode code points
- `Array`, which represents a numbered sequence of values
- `Object`, which represents an unordered map from strings to values
- `Asset`, which represents a blob
- `Archive`, which represents a map from strings to `Asset`s or `Archive`s
- `ResourceReference`, which represents a reference to a *Pulumi resource*
- `Unknown`, which represents a value whose type and concrete value are not known
- `Secret`, which demarcates a value whose contents are sensitive

Assets and Archives

An `Asset` or `Archive` may contain either literal data or a reference to a file or URL. In the former case, the literal data is a textual string or a map from strings to `Asset`s or `Archive`s, respectively. In the latter case, the referenced file or URL is an opaque blob or a TAR, gzipped TAR, or ZIP archive, respectively.

Each `Asset` or `Archive` also carries the SHA-256 hash of its contents. This hash can be used to uniquely identify the asset (e.g. for locally caching `Asset` or `Archive` contents).

Resource References

A `ResourceReference` represents a reference to a *Pulumi resource*. Although all that is necessary to uniquely identify a resource is its URN, a `ResourceReference` also carries the resource's ID (if it is a *custom resource*) and the version of the provider that manages the resource. If the contents of the referenced resource must be inspected, the reference must be resolved by invoking the `getResource` function of the engine's builtin provider. Note that this is only possible if there is a connection to the engine's resource monitor, e.g. within the scope of a call to `Construct`. This implies that resource references may not be resolved within calls to other provider methods. Therefore, configuration values, custom resources and provider functions should not rely on the ability to resolve resource references, and should instead treat resource references as either their ID (if present) or URN. If the ID is present and empty, it should be treated as an *`Unknown`*.

Unknowns

An `Unknown` represents a value whose type and concrete value are not known. Resources typically produce these values during *previews* for properties with values that cannot be determined until the resource is actually created or updated. *Functions* must not accept or return unknown values.

Secrets

A `Secret` represents a value whose contents are sensitive. Values of this type are merely wrappers around the sensitive value. A provider should take care not to leak a secret value, and should wrap any resource output values that are always sensitive in a `Secret`. *Functions* must not accept or return secret values.

Property Paths

TODO: write this up

6.2 Schema

Each provider constitutes the implementation of a single Pulumi package. Each Pulumi package has an associated schema that describes the package's *configuration*, *resources*, *functions*, and data types. The schema is primarily used to facilitate programmatic generation of per-language SDKs for the Pulumi package, but is also used for importing resources, program code generation, and more. Schemas may be expressed using JSON or YAML, and must validate against the *metaschema*.

6.3 Provider Lifecycle

Clients of a provider (e.g. the Pulumi CLI) must obey the provider lifecycle. This lifecycle guarantees that a provider is configured before any resource operations are performed or provider functions are invoked. The lifecycle of a provider instance is described in brief below.

1. The user *looks up* the factory for a particular `(package, semver)` tuple and uses the factory to create a provider instance.
2. The user *configures* the provider instance with a particular configuration object.
3. The user performs resource operations and/or calls provider functions with the provider instance.
4. The user *shuts down* the provider instance.

Within the scope of a Pulumi stack, each provider instance has a corresponding provider resource. Provider resources are custom resources that are managed by the Pulumi engine, and obey the usual *custom resource lifecycle*. The `Check` and `Diff` methods for a provider resource are implemented using the `CheckConfig` and `DiffConfig` methods of the resource's provider instance. The latter is criticially important to the user experience: if `DiffConfig` indicates that the provider resource must be replaced, all of the custom resources managed by the provider resource will *also* be replaced. Thus, `DiffConfig` should only indicate that replacement is required if the provider's new configuration prevents it from managing resources associated with its old configuration.

6.3.1 Lookup

Before a provider can be used, it must be instantiated. Instatiating a provider requires a `(package, semver)` tuple, which is used to find an appropriate provider factory. The lookup process proceeds as follows:

- Let the best available factory `B` be empty
- For each available provider factory `F` with package name `package`:
  - If the `F`'s version is compatible with `semver`:
    - If `B` is empty or if `F`'s version is newer than `B`'s version, set `B` to `F`
- If `B` is empty, no compatible factory is available, and lookup fails

Within the context of the Pulumi CLI, the list of available factories is the list of installed resource plugins plus the builtin `pulumi` provider. The list of installed resource plugins can be viewed by running `pulumi plugin ls`.

Once an appropriate factory has been found, it is used to construct a provider instance.

6.3.2 Configuration

A provider may accept a set of configuration variables. After a provider is instantiated, the instance must be configured before it may be used, even if its set of configuration variables is empty. Configuration variables may be of *any type*. Because it has no connection to the Pulumi engine during configuration, a provider's configuration variables should not rely on the ability to resolve *resource references*.

In general, a provider's configuration variables define the set of resources it is able to manage: for example, the `aws` provider accepts the AWS region to use as a configuration variable, which prevents a particular instance of the provider from managing AWS resources in other regions. As noted in the *overview*, changes to a provider's configuration that prevent the provider from managing resources that were created with its old configuration should require that those resources are destroyed and recreated.

Provider configuration is performed in at most three steps:

1. *CheckConfig*, which validates configuration values and applies defaults computed by the provider. This step is only required when configuring a provider using user-supplied values, and can be skipped when using values that were previously processed by `CheckConfig`.
2. *DiffConfig*, which indicates whether or not the new configuration can be used to manage resources created with the old configuration. Note that this step is only applicable within contexts where new and old configuration exist (e.g. during a *preview* or *update* of a Pulumi stack).
3. *Configure*, which applies the inputs validated by `CheckConfig`.

CheckConfig

`CheckConfig` implements the semantics of a custom resource's *Check* method, with provider configuration in the place of resource inputs. Each call to `CheckConfig` is provided with the provider's prior checked configuration (if any) and the configuration supplied by the user. The provider may reject configuration values that do not conform to the provider's schema, and may apply default values that are not statically computable. The type of a computed default value for a property should agree with the property's schema.

DiffConfig

`DiffConfig` implements the semantics of a custom resource's *Diff* method, with provider configuration in the place of resource inputs and state. Each call to `DiffConfig` is provided with the provider's prior and current configuration. If there are any changes to the provider's configuration, those changes should be reflected in the result of `DiffConfig`. If there are changes to the configuration that make the provider unable to manage resources created using the prior configuration (e.g. changing an AWS provider instance's region), `DiffConfig` should indicate that the provider must be replaced. Because replacing a provider will require that all of the resources with which it is associated are *also* replaced, replacement semantics should be reserved for changes to configuration properties that are guaranteed to make old resources unmanagable (e.g. a change to an AWS access key should not require replacement, as the set of resources accesible via an access key is easily knowable).

Configure

`Configure` applies a set of checked configuration values to a provider instance. Within a call to `Configure`, a provider instance should use its configuration values to create appropriate SDK instances, check connectivity, etc. If configuration fails, the provider should return an error.

Parameters

- `inputs`: the configuration `Object` for the provider. This value may contain *Unknown* values if the provider is being configured during a *preview*. In this case, the provider should provide as much functionality as possible.

Results

None.

6.3.3 Shutdown

Once a client has finished using a resource provider, it must shut the provider down. A client requests that a provider shut down gracefully by calling its `SignalCancellation` method. In response to this method, a provider should cancel all outstanding resource operations and funtion calls. After calling `SignalCancellation`, the client calls `Close` to inform the provider that it should release any resources it holds.

`SignalCancellation` is advisory and non-blocking; it is up to the client to decide how long to wait after calling `SignalCancellation` to call `Close`. Typically, a provider should check for the cancellation signal while polling for completion of an operation. If cancelling while waiting for a create operation to be completed, then a "partial state" should be returned in the error to include the provider-created id.

6.4 Custom Resource Lifecycle

A custom resource has a well-defined lifecycle within the scope of a Pulumi stack. When a custom resource is registered by a Pulumi program, the Pulumi engine first determines whether the resource is being read, imported, or managed. Each of these operations involves a different interaction with the resource's provider.

If the resource is being read, the engine calls the resource's provider's *`Read`* method to fetch the resource's current state. This call to *`Read`* includes the resource's ID and any state provided by the user that may be necessary to read the resource.

If the resource is being imported, the engine first calls the provider's *`Read`* method to fetch the resource's current state and inputs. This call to *`Read`* only inclues the ID of the resource to import; that is, *any importable resource must be identifiable using its ID alone*. If the *`Read`* succeeds, the engine calls the provider's *`Check`* method with the inputs returned by *`Read`* and the inputs supplied by the user. If any of the inputs are invalid, the import fails. Finally, the engine calls the provider's *`Diff`* method with the inputs returned by *`Check`* and the state returned by *`Read`*. If the call to *`Diff`* indicates that there is no difference between the desired state described by the inputs and the actual state, the import succeeds. Otherwise, the import fails.

If the resource is being managed, the engine first looks up the last registered inputs and last refreshed state for the resource's URN. The engine then calls the resource's provider's *`Check`* method with the last registered inputs (if any) and the inputs supplied by the user. If any of the inputs are invalid, the registration fails. Otherwise, the engine decides which operations to perform on the resource based on the difference between the desired state described by its inputs and its actual state. If the resource does not exist (i.e. there is no last refereshed state for its URN), the engine calls the provider's *`Create`* method, which returns the ID and state of the created resource. If the resource does exist, the action taken depends on the differences (if any) between the desired and actual state of the resource.

If the resource does exist, the engine calls the provider's *`Diff`* method with the inputs returned from *`Check`*, the resource's ID, and the resource's last refreshed state. If the result of the call indicates that there is no difference between the desired and actual state, no operation is necessary. Otherwise, the resource is either updated (if *`Diff`* does not indicate that the resource must be replaced) or replaced (if *`Diff`* does indicate that the resource must be replaced).

To update a resource, the engine calls the provider's *`Update`* method with the inputs returned from *`Check`*, the resource's ID, and its last refreshed state. *`Update`* returns the new state of the resource. The resource's ID may not be changed by a call to *`Update`*.

To replace a resource, the engine first calls *`Check`* with an empty set of prior inputs and the inputs supplied with the resource's registration. If *`Check`* fails, the resource is not replaced. Otherwise, the inputs returned by this call to *`Check`* will be used to create the replacement resource. Next, the engine inspects the resource options supplied with the resource's registration and result of the call to *`Diff`* to determine whether the replacement can be created before the original resource is deleted. This order of operations is preferred when possible to avoid downtime due to the lag between the deletion of the current resource and creation of its replacement. If the replacement may be created before the original is deleted, the engine calls the provider's *`Create`* method with the re-checked inputs, then later calls *`Delete`* with the resource's ID and original state. If the resource must be deleted before its replacement can be created, the engine first deletes the transitive closure of resource that depend on the resource being replaced. Once these deletes have completed, the engine deletes the original resource by calling the provider's *`Delete`* method with the resource's ID and original state. Finally, the engine creates the replacement resource by calling *`Create`* with the re-checked inputs.

If a managed resource registered by a Pulumi program is not re-registered by the next successful execution of a Pulumi progam in the resource's stack, the engine deletes the resource by calling the resource's provider's *`Delete`* method with the resource's ID and last refereshed state.

The diagram below summarizes the custom resource lifecycle. Detailed descriptions of each resource operation follow.

6.4.1 Lifecycle Methods

Check

The `Check` method is responsible for validating the inputs to a resource. It may optionally apply default values for unspecified input properties that cannot reasonably be computed outside the provider (e.g. because they require access to the provider's internal data structures).

Parameters

- `urn`: the *URN* of the resource.
- `olds`: the last recorded input `Object` for the resource, if any. If present, these inputs must have been generated by a prior call to `Check` or *`Read`*. These inputs will never contain *`Unknowns`*.
- `news`: the new input `Object` for the resource. These inputs may have been provided by the user or generated by a call to *`Read`*, and may contain *`Unknowns`*.

Results

- `inputs`: the checked input `Object` for the resource with default values applied. The types of the properties in `inputs` should agree with the types of the resource's input properties as described in its (schema)[#schema]. If `news` contains *`Unknowns`*, `inputs` may contain *`Unknowns`*.
- `failures`: any validation failures present in the inputs. These failures should be constrained to type and range mismatches. A failure is a tuple of a *property path* and a failure reason.

Diff

The `Diff` method is responsible for calculating the differences between the actual and desired state of a resource as represented by its last recorded state and new input `Object` as returned from *`Check`* or *`Read`* and the logical operation necessary to reconcile the two (i.e. no operation, an `Update, or a` Replace`).

Parameters

- `urn`: the *URN* of the resource.
- `id`: the *ID* of the resource.
- `olds`: the last recorded state `Object` for the resource. This `Object` must have been generated by a call to `Create`, `Read`, or `Update`, and will never contain *`Unknowns`*.
- `news`: the current input `Object` for the resource as returned by *`Check`* or *`Read`*. This value may contain *`Unknowns`*.
- `ignoreChanges`: the set of *property paths* to treat as unchanged.

Results

- `detailedDiff`: the *detailed diff* between the resource's actual and desired state.
- `deleteBeforeReplace`: if true, the resource must be deleted before it is recreated. This flag is ignored if `detailedDiff` does not indicate that the resource needs to be replaced.
- `changes`: an enumeration that indicates whether the provider detected any changes, detected no changes, or does not support detailed diff detection. Providers should return `Some` for this value if there are any entries in `detailedDiff`; otherwise they should return `None` to indicate no difference. If a provider returns `Unknown` for this value, it is the responsibility of the client to determine whether or not differences exist by comparing the resource's last recorded *inputs* with its current inputs.

In addition, the following properties should be returned for compatibility with older clients:

- `replaceKeys`: the list of top-level input property names with changes that require that the resource be replaced.
- `stableKeys`: the list of top-level input property names that did not change and top-level output properties that are guaranteed not to change.
- `changedKeys`: the list of top-level input property names that changed.

If a provider is unable to compute a diff because its configuration contained *`Unknowns`*, it can return an error that indicates as such. The client should conservatively assume that the resource must be updated and warn the user.

Detailed Diffs

A detailed diff is a map from *property paths* to change kinds that describes the differences between the actual and desired state of a resource and the operations necessary to reconcile the two.

Each entry in a detailed diff has a change kind that describes how the value of and input property differs, whether or not the difference requires replacement, and which old value was used for determining the difference. The core change kinds are:

- `Add`, which denotes an `Object` property or `Array` element that was added
- `Update`, which denotes an `Object` property or `Array` element that was updated
- `Delete`, which denotes an `Object` property or `Array` element that was removed

Each of these core kinds is paramaterized on whether or not the change requires replacement and whether the old value of the property should was read from the resource's old input `Object` or old state `Object`.

TODO: the input/output flag is a bit clumsy, as it is the only part of the system that implies some correspondence between input and output `Object` schemas. It was chosen over an approach that used old/new values due in order to remove the possibility of a provider accidentally revealing a secret value as part of a diff. We should reconsider this approach if we can find an easy way to maintain secretness.

Create

The `Create` method is responsible for creating a new instance of a resource from an input `Object` and returning the resource's state `Object`. `Create` may be called during a *preview* in order to compute a hypothetical state `Object` without actually creating the resource, in which case the `preview` argument will be `true`.

Parameters

- `urn`: the *URN* of the resource.
- `news`: the input `Object` for the resource. This value must have been generated by a prior call to `Check`. If `preview` is true, this value may contain *`Unknown`* value; otherwise, it is guaranteed to be fully-known.
- `timeout`: the timeout for the create operation. If this value is `0`, the provider should apply the default creation timeout for the resource.
- `preview`: if true, the provider should calculate the state `Object` as accurately as it is able without actually creating the resource. Top-level properties that are present in the resource's *schema* but are omitted from its state `Object` should be treated as having the value *`Unknown`*. Nested properties with values that are not computable must be explicitly set to *`Unknown`*. If it is not possible to guarantee that the value produced by a preview will match the value that would be produced by actually creating the resource, the value should be left unknown.

Results

- `id`: the ID for the created resource. If `preview` is true, this value will be ignored.
- `state`: the new state `Object` for the resource. If `preview` is true, this value may contain *`Unknowns`*.

Update

The `Update` method is responsible for updating a resource in-place in order given its last recorded state `Object` and current input `Object`. `Update` may be called during a *preview* in order to compute a hypothetical state `Object` without actually updating the resource, in which case the `preview` argument will be `true`.

Parameters

- `urn`: the *URN* of the resource.
- `id`: the *ID* of the resource.
- `olds`: the last recorded state `Object` for the resource. This `Object` must have been generated by a call to `Create`, `Read`, or `Update`.
- `news`: the input `Object` for the resource. This value must have been generated by a prior call to `Check`. If `preview` is true, this value may contain *`Unknown`* value; otherwise, it is guaranteed to be fully-known.
- `timeout`: the timeout for the update operation. If this value is `0`, the provider should apply the default update timeout for the resource.
- `ignoreChanges`: the set of *property paths* to treat as unchanged.

- `preview`: if true, the provider should calculate the state `Object` as accurately as it is able without actually updating the resource. Top-level properties that are present in the resource's *schema* but are omitted from its state `Object` should be treated as having the value *`Unknown`*. Nested properties with values that are not computable must be explicitly set to *`Unknown`*. If it is not possible to guarantee that the value produced by a preview will match the value that would be produced by actually updating the resource, the value should be left unknown.

Results

- `state`: the new state `Object` for the resource. If `preview` is true, this value may contain *`Unknowns`*.

Read

The `Read` method is responsible for reading the current inputs and state `Objects` for a resource. `Read` may be called during a *refresh* or *import* of a managed resource or during a *preview* or *update* for an external resource.

Parameters

- `urn`: the *URN* of the resource.
- `id`: the *ID* of the resource.
- `inputs`: the last recoded input `Object` for the resource, if any. If present, this `Object` must have been generated by a call to `Check` or `Read`. This parameter is omitted if the resource is being *imported*.
- `state`: the last recorded state `Object` for the resource, if any. This `Object` must have been generated by a call to `Create`, `Read`, or `Update`. This property is only present during a *refresh*, and must not be required for a resource to support *importing*.

Results

- `newInputs`: the new input `Object` for the resource. If the provider does not support *detailed diffs*, these inputs may be used by the engine to determine whether or not the resource's actual state differs from its desired state during the next *preview* or *update*. The shape of the returned `Object` should be compatible with the resource's *schema*. If the resource is being *imported*, an input `Object` must be returned. Otherwise, unless the input `Object` is used for computing default property values or the provider does not support *detailed diffs*, `newInputs` should simply reflect the value of `inputs`.
- `newState`: the new state `Object` for the resource.

Delete

The `Delete` method is responsible for deleting a resource given its ID and state `Object`.

Parameters

- `urn`: the *URN* of the resource.
- `id`: the *ID* of the resource.
- `state`: the last recorded state `Object` for the resource. This `Object` must have been generated by a call to `Create`, `Read`, or `Update`.
- `timeout`: the timeout for the delete operation. If this value is `0`, the provider should apply the default deletion timeout for the resource.

Results

None.

6.5 Component Resource Lifecycle

- TODO: user-level programming model

6.5.1 Construct

- TODO: brief, parameters, results, etc.

6.6 Provider Functions

6.6.1 Invoke

- TODO

6.6.2 StreamInvoke

- TODO

6.7 CLI Scenarios

- TODO:
  - preview
  - update
  - import
  - refresh
  - destroy

6.7.1 Preview

- TODO:
  - check
  - diff
  - create/update preview, read operation

6.7.2 Update

- TODO:
  - check
  - diff
  - create/update/read/delete operation

6.7.3 Import

- TODO: read operation

6.7.4 Refresh

- TODO: read operations

6.7.5 Destroy

- TODO: delete operation

6.8 Appendix

6.8.1 Out-of-Process Plugin Lifecycle

6.8.2 gRPC Interface

- TODO:
  - feature negotiation
  - data representation

CHAPTER

SEVEN

PULUMI PACKAGE METASCHEMA

A description of the schema for a Pulumi Package

`object`

7.1 Properties

7.1.1 `attribution`

Freeform text attribution of derived work, if required.

`string`

7.1.2 `config`

The package's configuration variables.

`object`

Properties

`required`

A list of the names of the package's required configuration variables.

`array`

Items: `string`

`variables`

A map from variable name to propertySpec that describes a package's configuration variables.

`object`

Additional properties: *Property Definition*

7.1.3 `description`

The description of the package. Descriptions are interpreted as Markdown.

`string`

7.1.4 `displayName`

The human-friendly name of the package.

`string`

7.1.5 `functions`

A map from token to functionSpec that describes the set of functions defined by this package.

`object`

Property names: *Token*

Additional properties: *Function Definition*

7.1.6 `homepage`

The package's homepage.

`string`

7.1.7 `keywords`

The list of keywords that are associated with the package, if any.

`array`

Items: `string`

7.1.8 `language`

Additional language-specific data about the package.

`object`

7.1.9 `license`

The name of the license used for the package's contents.

`string`

7.1.10 `logoUrl`

The URL of the package's logo, if any.

`string`

7.1.11 `meta`

Format metadata about this package.

`object`

Properties

`moduleFormat` (*required*)

A regex that is used by the importer to extract a module name from the module portion of a type token. Packages that use the module format "namespace1/namespace2/…/namespaceN" do not need to specify a format. The regex must define one capturing group that contains the module name, which must be formatted as "namespace1/namespace2/…namespaceN".

`string`

Format: `regex`

7.1.12 `name` (*required*)

The unqualified name of the package (e.g. “aws”, “azure”, “gcp”, “kubernetes”, “random”)

`string`

Pattern: `^[a-zA-Z][-a-zA-Z0-9_]*$`

7.1.13 `pluginDownloadUrl`

The URL to use when downloading the provider plugin binary.

`string`

7.1.14 `provider`

The provider type for this package.

Resource Definition

7.1.15 `publisher`

The name of the person or organization that authored and published the package.

`string`

7.1.16 `repository`

The URL at which the package’s sources can be found.

`string`

7.1.17 `resources`

A map from type token to resourceSpec that describes the set of resources and components defined by this package.

`object`

Property names: *Token*

Additional properties: *Resource Definition*

7.1.18 `types`

A map from type token to complexTypeSpec that describes the set of complex types (i.e. object, enum) defined by this package.

`object`

Property names: *Token*

Additional properties: *Type Definition*

7.1.19 `version`

The version of the package. The version must be valid semver.

`string`

Pattern: `^v?(?P<major>0|[1-9]\d*)\.(?P<minor>0|[1-9]\d*)\.(?P<patch>0|[1-9]\d*)(?:-(?P<prerelease>(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*))*))?(?:\+(?P<buildmetadata>[0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?$`

7.2 Alias Definition

`object`

7.2.1 Properties

`name`

The name portion of the alias, if any

`string`

`project`

The project portion of the alias, if any

`string`

`type`

The type portion of the alias, if any

`string`

7.3 Array Type

A reference to an array type. The "type" property must be set to "array" and the "items" property must be present. No other properties may be present.

`object`

7.3.1 Properties

`items` (*required*)

The element type of the array

Type Reference

`type` (*required*)

Constant: `"array"`

7.4 Enum Type Definition

Describes an enum type

`object`

7.4.1 Properties

`enum` (*required*)

The list of possible values for the enum

`array`

Items: *Enum Value Definition*

`type` (*required*)

The underlying primitive type of the enum

`string`

Enum: `"boolean"` | `"integer"` | `"number"` | `"string"`

7.5 Enum Value Definition

`object`

7.5.1 Properties

`deprecationMessage`

Indicates whether the value is deprecated.

`string`

`description`

The description of the enum value, if any. Interpreted as Markdown.

`string`

`name`

If present, overrides the name of the enum value that would usually be derived from the value.

`string`

`value` (*required*)

The enum value itself

`boolean` | `integer` | `number` | `string`

7.6 Function Definition

Describes a function.

`object`

7.6.1 Properties

`deprecationMessage`

Indicates whether the function is deprecated

`string`

`description`

The description of the function, if any. Interpreted as Markdown.

`string`

`inputs`

The bag of input values for the function, if any.

Object Type Details

`isOverlay`

Indicates that the implementation of the function should not be generated from the schema, and is instead provided out-of-band by the package author

`boolean`

`language`

Additional language-specific data about the function.

`object`

`outputs`

The bag of output values for the function, if any.

Object Type Details

7.7 Map Type

A reference to a map type. The “type” property must be set to “object” and the “additionalProperties” property may be present. No other properties may be present.

`object`

7.7.1 Properties

`additionalProperties`

The element type of the map. Defaults to “string” when omitted.

Type Reference

`type` (*required*)

Constant: `"object"`

7.8 Named Type

A reference to a type in this or another document. The “$ref” property must be present. The “type” property is ignored if it is present. No other properties may be present.

`object`

7.8.1 Properties

`$ref` (*required*)

The URI of the referenced type. For example, the built-in Archive, Asset, and Any types are referenced as “pulumi.json#/Archive”, “pulumi.json#/Asset”, and “pulumi.json#/Any”, respectively. A type from this document is referenced as “#/types/pulumi:type:token”. A type from another document is referenced as “path#/types/pulumi:type:token”, where path is of the form: “/provider/vX.Y.Z/schema.json” or “pulumi.json” or “http[s]://example.com/provider/vX.Y.Z/schema.json” A resource from this document is referenced as “#/resources/pulumi:type:token”. A resource from another document is referenced as “path#/resources/pulumi:type:token”, where path is of the form: “/provider/vX.Y.Z/schema.json” or “pulumi.json” or “http[s]://example.com/provider/vX.Y.Z/schema.json”

`string`

Format: `uri-reference`

`type`

ignored; present for compatibility with existing schemas

`string`

7.9 Object Type Definition

`object`

All of:

- *Object Type Details*

7.9.1 Properties

`type`

Constant: `"object"`

7.10 Object Type Details

Describes an object type

`object`

7.10.1 Properties

`properties`

A map from property name to propertySpec that describes the object’s properties.

`object`

Additional properties: *Property Definition*

`required`

A list of the names of an object type's required properties. These properties must be set for inputs and will always be set for outputs.

`array`

Items: `string`

7.11 Primitive Type

A reference to a primitive type. A primitive type must have only the "type" property set.

`object`

7.11.1 Properties

`type` (*required*)

The primitive type, if any

`string`

Enum: `"boolean"` | `"integer"` | `"number"` | `"string"`

7.12 Property Definition

Describes an object or resource property

`object`

All of:

- *Type Reference*

7.12.1 Properties

`const`

The constant value for the property, if any. The type of the value must be assignable to the type of the property.

`boolean` | `number` | `string`

`default`

The default value for the property, if any. The type of the value must be assignable to the type of the property.

`boolean` | `number` | `string`

`defaultInfo`

Additional information about the property's default value, if any.

`object`

Properties

`environment` (*required*)

A set of environment variables to probe for a default value.

`array`

Items: `string`

`language`

Additional language-specific data about the default value.

`object`

`deprecationMessage`

Indicates whether the property is deprecated

`string`

`description`

The description of the property, if any. Interpreted as Markdown.

`string`

`language`

Additional language-specific data about the property.

`object`

`replaceOnChanges`

Specifies whether a change to the property causes its containing resource to be replaced instead of updated (default false).

`boolean`

`willReplaceOnChanges`

Indicates that the provider will replace the resource when this property is changed.

`boolean`

`secret`

Specifies whether the property is secret (default false).

`boolean`

7.13 Resource Definition

Describes a resource or component.

`object`

All of:

- *Object Type Details*

7.13.1 Properties

`aliases`

The list of aliases for the resource.

`array`

Items: *Alias Definition*

`deprecationMessage`

Indicates whether the resource is deprecated

`string`

`description`

The description of the resource, if any. Interpreted as Markdown.

`string`

`inputProperties`

A map from property name to propertySpec that describes the resource's input properties.

`object`

Additional properties: *Property Definition*

`isComponent`

Indicates whether the resource is a component.

`boolean`

`isOverlay`

Indicates that the implementation of the resource should not be generated from the schema, and is instead provided out-of-band by the package author

`boolean`

`methods`

A map from method name to function token that describes the resource's method set.

`object`

Additional properties: `string`

`requiredInputs`

A list of the names of the resource's required input properties.

`array`

Items: `string`

`stateInputs`

An optional objectTypeSpec that describes additional inputs that mau be necessary to get an existing resource. If this is unset, only an ID is necessary.

Object Type Details

7.14 Token

`string`

Pattern: `^[a-zA-Z][-a-zA-Z0-9_]*:([^0-9][a-zA-Z0-9._/-]*)?:[^0-9][a-zA-Z0-9._/]*$`

7.15 Type Definition

Describes an object or enum type.

`object`

One of:

7.15.1 Properties

`description`

The description of the type, if any. Interpreted as Markdown.

`string`

`isOverlay`

Indicates that the implementation of the type should not be generated from the schema, and is instead provided out-of-band by the package author

`boolean`

`language`

Additional language-specific data about the type.

`object`

7.16 Type Reference

A reference to a type. The particular kind of type referenced is determined based on the contents of the "type" property and the presence or absence of the "additionalProperties", "items", "oneOf", and "$ref" properties.

`object`

One of:

7.16.1 Properties

`plain`

Indicates that when used as an input, this type does not accept eventual values.

`boolean`

7.17 Union Type

A reference to a union type. The "oneOf" property must be present. The union may additional specify an underlying primitive type via the "type" property and a discriminator via the "discriminator" property. No other properties may be present.

`object`

7.17.1 Properties

`discriminator`

Informs the consumer of an alternative schema based on the value associated with it

`object`

Properties

`mapping`

an optional object to hold mappings between payload values and schema names or references

`object`

Additional properties: `string`

`propertyName` (*required*)

PropertyName is the name of the property in the payload that will hold the discriminator value

`string`

`oneOf` (*required*)

If present, indicates that values of the type may be one of any of the listed types

`array`

Items: *Type Reference*

`type`

The underlying primitive type of the union, if any

`string`

Enum: `"boolean"` | `"integer"` | `"number"` | `"string"`

CHAPTER

EIGHT

BUILDING THE DOCS

This documentation is generated using Sphinx and authored in Markdown. Markdown support for Sphinx is provided by MyST. MyST provides a number of small syntax extensions to support declaring ReStructuredText directives; see the MyST syntax guide for details.

In order to build the devloper documentation:

1. Install PlantUML. On macOS, this can be done via `brew install plantuml`.
2. Install the requirements for Sphinx:

   ```
   $ pip install requirements.txt
   ```

3. Run `make` to build the HTML documentation:

   ```
   $ make
   ```

This will regenerate any out-of-date SVGs and build the a local version of the HTML documentation. The documentation can also be built from the repository root by running `make developer_docs`.

Note that Sphinx doesn't do a great job of rebuilding output files if only the table-of-contents has changed. If you change the table of contents, you may need to clean the output directory in order to see the effects of your changes:

~~~
```bash
$ make clean
```
~~~

8.1 Notes on Style

- Do use appropriate links wherever possible. Learn to use header anchors.
- If a particular link destination is referenced multiple times, prefer shortcut reference links.