Specification: Introduce lazy Configuration factory method APIs

Abstract

Introduce new factory methods on `ConfigurationContainer` allowing users to create role-locked `Configuration`s, where roles are specified upon construction.

Spike

Prior Work

Specification: Configuration Roles Changes in Gradle 8.3
Configuration Purposes Can Be Declared and Tracked

Background

The `Configuration` type is a pervasive, all-powerful type that currently supports three separate mutually exclusive use-cases:

- **Resolution**: Configurations can act as the root of a dependency graph and can initiate resolutions of dependency graphs. These graphs can be leveraged to resolve artifact sets.
- **Consumption**: Configurations can act as a variant in the context of Dependency Management and Publishing – advertising dependencies and artifacts to outside consumers.
- **Dependency Declaration**: Configurations can act as a named source of dependencies, dependency constraints, and exclude rules, for use with resolvable and consumable configurations.

The `Configuration` type already, to an extent, defines some separation in regards to these use cases by defining methods that allow setting and querying whether a given `Configuration` supports one or more of these given roles:

```java
interface Configuration {
    void setCanBeConsumed(boolean allowed);
    boolean isCanBeConsumed();
```

```
    void setCanBeResolved(boolean allowed);
    boolean isCanBeResolved();
    void setCanBeDeclared(boolean allowed);
    boolean isCanBeDeclared();
}
```

Implementing this mutable-style behavior is difficult, since at any time a `Configuration` may need to add entire new behaviors to the set which it currently supports. Furthermore, `Configuration`s upon construction support all three of these roles, while in most cases only needs to support a single role. Currently, a user would use the following code in order to create a “specialized” configuration which supports only a single role

Java
```
configurations {
    myDependencies {
        canBeConsumed = false
        canBeResolved = false
    }
    myResolvable {
        canBeConsumed = false
        canBeDeclared = false
        extendsFrom(myDependencies)
    }
    myConsumable {
        canBeResolved = false
        canBeDeclared = false
        extendsFrom(myDependencies)
    }
}
```

Why

- Since the current API only supports creating a Configuration which supports all three roles, new `Configuration`s need to be initialized in a state which can support all three roles. This means extra memory in terms of state and services need to be retained in

any given `Configuration` implementation upon creation – even if these behaviors will be turned off immediately after construction

- Even if a given usage is turned off immediately, the `Configuration` API will still present methods related to that turned-off configuration. For example, `apiElements` is a `Configuration`, which extends `FileCollection`, even though all methods on that `Configuration` which relate to resolution will throw exceptions when called.
- The mutability of roles make it difficult to configure all `Configuration`s of a single role, particularly resolvable `Configuration`s. Consider a use-case where a user wants to affect all resolutions in a single project, for example by adding a request attribute for use with artifact transforms or by configuring the resolution strategy for all configurations. Currently, any `configurations.all` callbacks will be executed before the configure block of a given `Configuration`, meaning that callback will be executed against the configuration *before* its roles are configured. This necessitates using an `afterEvaluate` in order to have a `configurations.all` properly determine the type of a `Configuration`. We officially recommend this pattern in our minifiy sample.
- The mutability of roles can further bring about confusing situations when a role for a given `Configuration` is disabled *after* the `Configuration` has been configured for a behavior of that role. Consider a case where the user configures the `ResolutionStrategy` for a `Configuration`, and then disables the resolvable usage for that configuration. What does that mean conceptually? It should not be possible for this situation to arise.
    - KMP does this and has stopped us from shipping features.

Given all of these reasons, the primary goal that this spec is trying to achieve is to allow users to specify `Configuration` roles upon creation.

- This avoids “multi-role” configurations, such as those which are resolvable and can be consumed
- This ensures a `Configuration` which intended to be one type cannot be changed to another
- This resolves the `configurations.all` requiring `afterEvaluate` issue.

Given the proposed change, we will be able to work towards the following future goals:

- Reduced `Configuration` memory footprint
    - If we know which role a `Configuration` is intended to perform, we can avoid allocating unnecessary state.
- Improved API
    - Future work will introduce new APIs which only expose the methods relevant to a given `Configuration` role. For example, a `Resolvable` will not have methods to add artifacts, a `Consumable` will not have methods to resolve it to a set of files, and a `DependencyScope` will not have resolution or consumption specific methods.

Functional design

We propose adding a series of new public and internal methods to `ConfigurationContainer` which *lazily* create `Configuration`s in specific roles. The `Configuration`s created by the public methods will be *locked*, meaning their roles will not be able to be changed after construction. The internal methods will create *unlocked* `Configuration`s, which may have their roles changed. In 9.0 we will migrate from the internal unlocked to public locked methods for first-party Gradle plugins.

Public factory methods

For each of the three configuration types, we will create two overloads with names `resolvable`, `consumable`, and `dependencyScope`, respectively. Each set of overloads will have a version that accepts a `String`, and another that accepts a `String` and an `Action` configure closure. We explicitly chose not to expose `maybeCreate` versions of these factory methods.

```
interface ConfigurationContainer {
    NamedDomainObjectProvider<ResolvableConfiguration> resolvable(String name);
    NamedDomainObjectProvider<ResolvableConfiguration> resolvable(String name, Action<? super ResolvableConfiguration> action);
    NamedDomainObjectProvider<ConsumableConfiguration> consumable(String name);
    NamedDomainObjectProvider<ConsumableConfiguration> consumable(String name, Action<? super ConsumableConfiguration> action);
    NamedDomainObjectProvider<DependenciesConfiguration> dependencyScope(String name);
    NamedDomainObjectProvider<DependenciesConfiguration> dependencyScope(String name, Action<? super DependenciesConfiguration> action);
}
```

Choosing a proper name for `dependencyScope` `Configuration`s has been difficult. Please see the Naming section below for details on the options we considered. Please add a reaction to the name which you think is best, or propose others if you think there are good names which we may have missed.

Internal factory methods

In order to avoid breaking changes, we cannot immediately update our first-party `Configuration`s to have locked roles. Therefore, we introduce internal methods to temporarily create unlocked `Configuration`s. The `Configuration`s created by the unlocked methods will emit deprecation warnings when their roles are changed, so by 9.0 we will be able to migrate first-party Gradle `Configuration`s to use the public API methods. These methods will have similar overloads compared to the public API, but will have the `Unlocked` suffix on their method names.

In addition to the three types of `Configuration`s that can be created with the public API, we also include two additional types in the internal API: `migrating`, and `resolvableDependencyScope`.

- `migrating` factory methods accept an additional `ConfigurationRole` parameter, which must be defined in the known set of `ConfigurationRolesForMigration`, and define `Configuration`s which have an arbitrary role now in 8.x but will migrate to a well-known role in 9.0. For example a Legacy (consumable, resolvable, dependencies) configuration migrating to a Resolvable.
- Resolvable + Dependency Scope `Configuration`s are used throughout our first-party plugins. We are still deciding how and if these types of `Configuration`s can be created by users. If we decide to expose this functionality to users, we will likely change the existing `create`, `maybeCreate`, and `register` methods to create these types of configurations. Defining public factory methods to create these types of `Configuration`s is out-of-scope for this document. Users who need this functionality can use the existing Legacy `Configuration` creating methods.

Finally, we introduce `maybeRegister` overloads for each configuration factory type. These are to be used internally and will emit deprecation warnings if the user creates a Gradle-managed configuration. By 9.0 we will be able to migrate away from the `maybe` factory methods.

```
interface RoleBasedConfigurationContainerInternal {
    NamedDomainObjectProvider<ConsumableConfiguration> consumableUnlocked(String name);
    NamedDomainObjectProvider<ConsumableConfiguration> consumableUnlocked(String name, Action<? super ConsumableConfiguration> action);
    NamedDomainObjectProvider<ResolvableConfiguration> resolvableUnlocked(String name);
    NamedDomainObjectProvider<ResolvableConfiguration> resolvableUnlocked(String name, Action<? super ResolvableConfiguration> action);
    NamedDomainObjectProvider<DependenciesConfiguration> dependencyScopeUnlocked(String name);
    NamedDomainObjectProvider<DependenciesConfiguration> dependencyScopeUnlocked(String name, Action<? super DependenciesConfiguration> action);
    NamedDomainObjectProvider<Configuration> migratingUnlocked(String name, ConfigurationRole role);
    NamedDomainObjectProvider<Configuration> migratingUnlocked(String name, ConfigurationRole role, Action<? super Configuration> action);
    NamedDomainObjectProvider<Configuration> resolvableDependencyScopeUnlocked(String name);
    NamedDomainObjectProvider<Configuration> resolvableDependencyScopeUnlocked(String name, Action<? super Configuration> action);
    NamedDomainObjectProvider<? extends Configuration> maybeRegisterResolvableUnlocked(String name, Action<? super Configuration> action);
    NamedDomainObjectProvider<? extends Configuration> maybeRegisterConsumableUnlocked(String name, Action<? super Configuration> action);
    NamedDomainObjectProvider<? extends Configuration> maybeRegisterDependencyScopeUnlocked(String name, Action<? super Configuration> action);
    NamedDomainObjectProvider<? extends Configuration> maybeRegisterDependencyScopeUnlocked(String name, boolean warnOnDuplicate, Action<? super Configuration> action);
    NamedDomainObjectProvider<? extends Configuration> maybeRegisterMigratingUnlocked(String name, ConfigurationRole role, Action<? super Configuration> action);
    NamedDomainObjectProvider<? extends Configuration> maybeRegisterResolvableDependencyScopeUnlocked(String name, Action<? super Configuration> action);
}
```

New configuration sub-interfaces

The public locked factory methods will return sub-types of `Configuration` corresponding to the three types of `Configurations` noted above:

- `ResolvableConfiguration`
- `ConsumableConfiguration`
- `DependencyScopeConfiguration`

With these new sub-interfaces, users will be able to more easily configure Configurations of a specific type in a declarative manner:

```
Java
configurations {
    withType(ResolvableConfiguration).configureEach {
        resolutionStrategy {
            // ...
        }
    }
}
```

Currently, these interfaces will be marker-only and have no methods, however future work will additional methods to expose these configurations through a new API. This API will have types which expose only role-specific functionality. For example, a `Resolvable` will be able to produce a `FileCollection`, `ArtifactCollection`, `ArtifactView`s, and any other behavior which a resolvable configuration is expected to do. The same goes for new `Consumable` and `DependencyScope` types.

In order to expose these new types, we will add a corresponding `toX` method on each `Configuration` subtype. For example, a `ResolvableConfiguration` would have a `toResolvable` method on it, `ConsumableConfiguration` will have a `toConsumable` method, and a `DependencyScopeConfiguration` will have a `toDependencyScope` method. These methods provide a bridge between the existing `Configuration` API and any new types which we introduce as part of a migration away from the `Configuration` type.

While there is no concrete timeline for the introduction of the new `Resolvable`, `Consumable`, and `DependencyScope` types, we are actively working towards a design in this area. Stay tuned for future updates here.

Implementing the new interfaces

To implement these new interfaces, we create 3 new concrete sub-classes of `Configuration`, each of which extends `DefaultConfiguration`, and implement any associated `Configuration` sub-interfaces. The new implementations include:

- `DefaultResolvableConfiguration`
- `DefaultConsumableConfiguration`
- `DefaultDependencyScopeConfiguration`

Each of these implementations will be *locked*, meaning they cannot change roles. This gives us the freedom to iteratively override methods on a per-role basis – for example throwing exceptions when a method is called on a `Configuration` with the incorrect role. For example, `getFiles()` on a `DefaultConsumableConfiguration` could throw an exception outright as opposed to first verifying its role like `DefaultConfiguration` does now. This greatly simplifies the implementation of `Configuration` by allowing us to avoid tracking the current role.

The existing Configuration factories

The Groovy DSL allows creating “legacy” configurations through the existing means. For example:

```
Unset
// Groovy
configurations {
    conf { }
}

// Kotlin
configurations {
    "conf" { }
}

val anotherConf by configurations.creating

// Java
Configuration conf = configurations.create("name")
```

To implement these existing factory schemes, we further define a `DefaultUnlockedConfiguration`, which is the type used for all unlocked configurations. Since the roles of these configurations *can* change, it would not be correct for it to implement any of the new proposed `Configuration` sub-interfaces . Instead, as configurations migrate

over to the factory methods and have locked roles, they will begin to be able to expose the new configuration subtypes.

In addition, `DefaultConfiguration` will be made `abstract`. Assuming we do not run into any blockers with third-party plugins using internal APIs, we will then rename `DefaultConfiguration` to `AbstractConfiguration`.

Future Work

For the time being, the “legacy” factories will not change in behavior. The linked spike does not implement the below features. The will continue to make *unlocked* `Configuration`s which default to the “legacy” state of having all 3 roles. These existing factory schemes are in use heavily within buildscripts and we want to avoid large changes to source for now. Due to the heavy use of these methods, we will want any deprecation strategy to be as seamless as possible.

We will likely start by deprecating these types of configurations for consumption. The `consumable` factory method will act as a replacement. This leaves the resolvable and dependency spec roles still enabled, solving the use-case of convenient in-buildscript downloading of arbitrary files from Maven repositories.

Next, we will deprecate these configurations from changing roles. This provides us with similar advantages that the new proposed factory methods provide.

However, these is no timeline for these changes. The greatest blocker for moving forwards here is that these APIs are heavily used in third-party plugins like Android and Kotlin, and we want to be sure they are not using deprecated behavior before continuing. We will likely need to support other popular plugins as part of this migration in order to ensure the change is as seamless as possible.

Outcome

After these changes, the code-block in Background section would now appear like so:

```java
configurations {
    dependencyScope("myDependencies")
    resolvable("myResolvable") {
        extendsFrom(myDependencies)
    }
    consumable("myConsumable") {
```

```
        extendsFrom(myDependencies)
    }
}
```

Questions/problems

- How should we handle `Configuration`s which are currently Resolvable + Dependency Scope `Configuration`s in the public API? It is a very common use-case for a user to add dependencies to a configuration just to resolve it immediately – often to fetch some arbitrary file from a Maven repository for further processing in an arbitrary task.
    - We use these types of configurations often. `classpath`, `annotationProcessor`, `antlr`, `pmd`, `checkstyle`, `codenarc`, `jacocoAgent`, `jacocoAnt`, `cppCompile`, `nativeLink`, `swiftCompile`, `swiftLink`, `providedCompile`, `providedRuntime`, `scalaCompilerPlugins`, `zinc`, and soon to be `earlib` and `deploy`.
    - Detached configurations will become a Resolvable + Dependencies configuration in 9.0
- Should the `NamedDomainObjectProvider`s returned by factory methods have `? extends` for their generic type? For example, for a `ResolvableConfiguration`, which should we return:
    - `NamedDomainObjectProvider<ResolvableConfiguration>`
    - `NamedDomainObjectProvider<? extends ResolvableConfiguration>`

Naming

Configurations used for declaring dependencies, dependency constraints, and exclude rules have gone by several names in the past, however none has significantly stuck. Furthermore, the Gradle documentation does not refer to these types of configurations with a consistent name. Below are the options we considered for this type of configuration.

- Bucket
    - This term is highly overloaded. See this, this, and this for examples of other buckets.
    - Bucket does not in any way describe what this type of configuration does, other than *collecting stuff*.
- Dependencies

    - This now at least has “dependency” in the name, but is a bit too general. The factory method within the configurations block (`dependencies`) would look very similar to the top-level dependencies block on the Project
- Dependency Spec
    - This is better, though its name makes it seem like it is a Spec which applies to a single dependency
- Dependencies Spec
    - Solves the above problem, but the double S is a tongue-twister / unwieldy to pronounce
- Dependency Scope
    - Uses familiar terminology borrowed from Maven
    - The “scope” term matches the type it is describing – the “Dependency scope configuration” represents the scope or extent as to where those dependencies will be used
    - Can potentially be confused with other overloads of scopes, for example variable scoping, though given this is explicitly prefixed with `Dependency` and since this term is already heavily used in the ecosystem with Maven, this is not much of an issue.
- DependencyDeclarations

Security implications

N/A

Considered Alternatives

- Introduce separate containers which expose new interfaces
    - `Configuration` is a highly-used type so we need a stepping-stone to the new interfaces in the interim.
    - We should not add new global containers to the `Project`
- Factory methods return new interfaces altogether instead of interfaces that extend `Configuration`
    - We want users to be able to easily adopt these new factory methods, since the primary goal initially is to migrate users to `Configuration`s which do not change roles.
    - The new interfaces are not finalized or even designed yet. Waiting for this would push back our goals of normalizing locked-role configurations
- Factory methods return `Configuration` and not subtypes of `Configuration`
    - This would mean adding `toResolvable` etc to the `Configuration` type itself. This would be a bad API since for example, a non-resolvable API would need to throw an exception for this method.

- Internally segment the `Configuration` type into separate implementations and use a sort of state-machine implementation to delegate to each underlying implementation based on the current role
    - This does not solve the problem of a Configuration itself exposing APIs for separate usages. Even if we could separate the implementations for each usage, we would still expose the mixed-usage APIs which is confusing from a user-experience standpoint.
- Do nothing
    - This is similar to the first bullet.
    - Since the `Configuration` API is so widely used, our initial goals should be to introduce an API that users can easily migrate to while also allowing users to leverage future upcoming interfaces in a type-safe manner

Won't Do

- Deprecate *all* configurations from changing roles
    - We want to do this, but it is likely we cannot migrate the entire ecosystem over to role-locked configurations by Gradle 9. Legacy configurations created with `create`, `maybeCreate`, and `register` will still be allowed to change roles. We will likely deprecate creating legacy configurations in 9.1
- Introduce new `Resolvable`, `Consumable`, and `DependencySpec` interfaces
    - This will be done in future proposals
- Split up and simplify `DefaultConfiguration`
    - Until `Configuration`s are limited from changing roles completely, we must always have an implementation of `Configuration` which supports all three usages.
    - If we updated `create`, `maybeCreate`, and `register` to only create a Resolvable + Dependencies configuration, we could potentially after 9.0 remove the Consumable functionality out of `DefaultConfiguration`.
- Implement specific methods in `Configuration` sub-implementations (for example hard-code `false` for `isCanBeConsumed` or hard-code throwing an exception for resolvable methods in `DefaultConsumableConfiguration`)
    - A proper implementation of `DefaultConsumableConfiguration` would be able to override non-consumable methods, for example `getIncoming` to throw an exception immediately.
    - This proposal does open the door for doing this, which we plan to do in future work.