KIP-421: Automatically resolve external configurations.

Status

Current state: accepted

Discussion thread: *here*

JIRA: *here*

Motivation

Kafka's ConfigDef is used for specifying the set of expected configurations throughout the clients, Kafka Connect, and Kafka Streams. For each configuration, you can specify the name, the type, the default value, the documentation, the group information, the order in the group, the width of the configuration value, the name suitable for display in the UI, validation logic the user may provide to perform single configuration validation, recommenders to get valid values for a configuration given the current configuration values etc. Kafka's AbstractConfig class is a convenient base class for components to extend and use to parse, validate, and access configurations that adhere to a specified ConfigDef. The AbstractConfig class holds both the original configuration that was provided as well as the parsed configurations.

KIP-297 was recently introduced to provide a way to use *variables* within configuration files and a ConfigProvider extension to replace these variables with values found outside the configuration file. This mechanism allows Connect, for example, to store secrets outside of a connector configuration and to insert those externally stored secrets into the configuration just prior to usage. However, components that use the KIP-297 ConfigProviders must manage those providers and explicitly replace the variables.

This proposal intends to enhance the AbstractConfig base class to automatically resolve variables of the form specified in KIP-297. This simple change will make it very easy for all the components client applications, Kafka Connect, and Kafka Streams to use shared code to easily incorporate externalized secrets and other variable replacements within their configurations. All the components broker, connect, producer, consumer, admin client, and so forth. will be able to automatically resolve the external configurations. This proposal does not add other ConfigProvider implementations or change the behavior of existing methods.

Public Interfaces

This proposal will add a new constructor to the AbstractConfig base class :
AbstractConfig

```
/**
 * Construct a configuration with a ConfigDef, the configuration properties, optional {@link ConfigProviders}.
 * @param definition the definition of the configurations
 * @param originals the name-value pairs of the configuration
 * @param configProviders the map of properties of config providers which will be instantiated by the
constructor to resolve any variables in {@code originals}
 * @param doLog whether the configurations should be logged
 */
public AbstractConfig(ConfigDef definition, Map<?, ?> originals, Map<?, ?> configProviders, boolean doLog) {
...
}
```

The KIP proposes on using the existing AbstractConfig to automatically resolve indirect variables. The `originals` configurations will contain both the config provider configs as well as configuration properties. The constructor will first instantiate the ConfigProviders using the config provider configs, then it will find all the variables in the values of the `originals` configurations, attempt to resolve the variables using the named ConfigProviders, and then do the normal parsing and validation of the configurations. If ConfigProvider is not provided in the originals, the constructor will skip the variable substitution step and will simply validate and parse the supplied configuration. With this approach both the configs and config providers can be defined in the same config property file.

Variables are defined by KIP-297 and have the form "`${providerName:[path:]key}`", where "providerName" is the name of a ConfigProvider, "path" is an optional string, and "key" is a required string. Per KIP-297, this variable is resolved by passing the "key" and optional "path" to a ConfigProvider with the specified name, and the result from the ConfigProvider is then used in place of the variable. Variables that cannot be resolved by the AbstractConfig constructor will be left unchanged in the configuration.

We propose to add new constructor which is similar to the existing constructor with an additional field Map<?, ?> configProviders to specify the config providers to use for resolving the configs. This constructor will provide a way to support components which do not have configProviders defined in the config file but read it from a separate config provider file. This constructor will first instantiate the ConfigProviders using the map of config provider and then resolve the variable as previous constructor. Any providers in originals map will be ignored.

Configuration Changes

This KIP proposes to reserve all config names starting with "config.providers" to resolve the config values. The config "config.providers" will be reserved to list the ConfigProviders, "`config.providers.<providerName>.class" will be reserved to specify` the class to be used to instantiate the providers and "`config.providers.<providerName>.<variableName>`" to specify any additional configs required by providers where `<providerName> is the name of the new config provider and <variableName> are the variables required by the config provider`. The new constructor will look for "config.providers" in the originals fields and instantiate them using the specified configProvider class. It will then use these instantiated config providers to resolve any indirect values.

Proposed Changes

With this KIP all the components broker, connect, producer, consumer, admin client, and so forth. to will be able to automatically resolve the external configurations. As this feature is built upon KIP-297, each component will have to make the below changes to add externalized configurations

1. Identify the configs which need to be stored in external systems.
2. Implement or use an existing ConfigProvider which will fetch the config value from external system.
3. Replace the configs values with variables as defined by KIP-297

Once these variables are added, the components will automatically get the resolved configurations via subclasses of AbstractConfig.

Dynamic configurations for Brokers

In order to support automatic resolution of indirect dynamic configurations, the brokers will have to reload the configurations when the config values are updated. This can be achieved by updating the broker using the AdminClient and kafka.sh script as described by KIP-226. The dynamic config updates need to happen atomically in order to achieve this the user will have to first update the external configs after all the external configs are updated, he will send a AdminClient request to the broker to update those config from config provider. Upon receiving the alterConfig request, the broker will refetch the updated configs from the ConfigProvider. The actual resolution of indirect values will be done by the KafkaConfig which extends from AbstractConfig, it will invoke the ConfigProvider to provide the actual values for the updated configs.

Example:

Consider the dynamic configurations ssl.keystore.location, ssl.keystore.password and ssl.keystore.type are stored in vault and their values are resolved using a VaultConfigProvider.

```
...
ssl.keystore.location=${vault:/path/to/variables.properties:ssl.keystore.location}

ssl.keystore.password=${vault:/path/to/variables.properties:ssl.keystore.password}

ssl.keystore.type=${vault:/path/to/variables.properties:ssl.keystore.type}

config.providers=vault
config.providers.file.class=org.apache.kafka.connect.configs.VaultConfigProvider
```

- The user will update the ssl.keystore.location, ssl.keystore.password, ssl.keystore.type in the vault.
- After the updates are complete he will send a adminClient request to the broker to notify that configs are updated.
- Once the Broker receives a alteredConfig request it will invoke the get function in VaultConfigProvider.
- The VaultConfigProvider will fetch the actual values for ssl.keystore.location, ssl.keystore.password, ssl.keystore.type from the vault.
- The broker will validate these configs and apply the changes.

Example

The following is an example config file that defines the configs and config providers in the same file and another notional ConfigProvider implementation named "vault":
Example of ConfigValues

```
...
foo.baz=/usr/temp/
foo.bar=${file:/path/to/variables.properties:foo.bar}

config.providers=file,vault
config.providers.file.class=org.apache.kafka.connect.configs.FileConfigProvider
config.providers.file.other.prop=another value passed to the class
config.providers.vault.class=com.acme.configs.CustomVaultConfigProvider
config.providers.vault.host=XYZ
config.providers.vault.location="/usr/location"
```

The first `foo.baz` property is a typical name-value pair commonly used in all Kafka configuration files. The `foo.bar` property has a value that is a KIP-297 *variable* of the form "`${providerName:[path:]key}`", where "providerName" is the name of a ConfigProvider, "path" is an optional string, and "key" is a required string. Per KIP-297, this variable is resolved by passing the "foo.bar" key and "/path/to/variables.properties" path to a ConfigProvider with the name "file", and the string returned from the ConfigProvider is then used in place of the variable.

An application or component can use or create a custom subclass of the AbstractConfig, and use this class to parse and validate the configurations in a configuration file. If the application or component does not provide any ConfigProviders, then the AbstractConfig class will simply parse the configuration as before without resolving the variable. If, however, the component or application does pass a config properties for ConfigProviders to the AbstractConfig constructor, the constructor will attempt to instantiate the ConfigProvider instance and use it to resolve and replace the `foo.bar` variable value prior to parsing and validating the configuration.

Example use of the existing AbstractConfig constructor

```
// Define the configurations
ConfigDef configDef = ...

// Parse and validate the configuration, using the ConfigProviders to resolve any variables
AbstractConfig config = new AbstractConfig(configDef, props, true);
```

This constructor will first instantiate the FileConfigProvider using the `props` configuration properties, and then attempt to resolve the `${file:/path/to/variables.properties:foo.bar}` variable with the `FileConfigProvider` instance by making a call similar to `fileConfigProvider.get("/path/to/variables.properties", Collections.singletonSet("foo.bar")`, and using the result as the new value for the `foo.bar` property in the configuration. The constructor will then parse and validate the configuration using the supplied ConfigDef as normal.

Example of the new AbstractConfig constructor with separate configProvider

```
// Define the configurations
ConfigDef configDef = ...

// Define the configProviders // This can be read from a separate properties file
Map<?,?> props = new HashMap<String, String>()
props.put("config.providers", "file")
props.put("config.providers.file.class","org.apache.kafka.connect.configs.FileConfigProvider")

// Parse and validate the configuration, using the ConfigProviders to resolve any variables
AbstractConfig config = new AbstractConfig(configDef, props, configProviders, true);
```

This new constructor will first instantiate the ConfigProviders passed in the configProviders object and then look for any variables in the supplied `props` object, find the `${file:/path/to/variables.properties:foo.bar}` variable and attempt to resolve it with the `FileConfigProvider` instance by making a call similar to `fileConfigProvider.get("/path/to/variables.properties", Collections.singletonSet("foo.bar")`, and using the result as the new value for the `foo.bar` property in the configuration. If any configProviders are passed in props object they will be ignored. The constructor will then parse and validate the configuration using the supplied ConfigDef as normal.

Compatibility, Deprecation, and Migration Plan

- This KIP proposes to reserve all config names starting with "config.providers" to resolve the config values. The config "config.providers" will be reserved to list the ConfigProviders, "`config.providers.<providerName>.class" will be reserved to specify` the class to be used to instantiate the providers and "`config.providers.<providerName>.<variableName>"` to specify any additional configs required by providers. If any current configs match this custom configs format and auto resolution is enabled, we will try to resolve the value to a actual value by fetching it from the config provider, If a matching key is found in the config provider it will replace its value and could result in a failure. In order to prevent such side effects we will update any existing configs matching the format.

Rejected Alternatives

Several other designs were considered but ultimately rejected.

1. Assume the ConfigProvider instances are also configured in the same configuration provided to the AbstractConfig constructor. This was rejected because it did not provide enough flexibility. For example, in Kafka Connect, the worker configurations can define ConfigProviders, whereas the connector configurations do not. Assuming the ConfigProviders are defined in the connector configuration would change the behavior of Connect.
2. Place the new logic in a separate class or as static methods. This is a viable option, but the current design was thought to be a bit more usable.
3. In order to support dynamic configs add a new DynamicConfiguration class that allows a component to obtain an initial configuration instance (subclass of AbstractConfig) and to add a listener that will be called when any of the configuration properties resolved from ConfigProviders change in the underlying provider. This was rejected as it does not handle atomic updates efficiently which is required by broker configs.
4. In order to support atomic dynamic configs enhance ConfigProvider to handle batch updates of configs by adding few seconds of delay before the fetching the actual values and modify ConfigChangeCallback to pass all the config changes at once in one giant map. This is not a viable option as related configs may be obtained from different ConfigProviders and adding delay in the config provider won't guarantee atomic updates.
5. Add a new constructor to AbstractConfig with a flag to enable/disable the feature by the subclass of AbstractConfig. This approach was rejected as the user won't be able to enable/disable the feature using configs or cli and feature should be enabled by default for all the components.