# KIP-458: Connector Client Config Override Policy



## Status

**Current state**: Accepted

**Discussion thread**: *here*

**Vote thread**: *here*

**JIRA**: Unable to render Jira issues macro, execution error.

**Release:** *AK 2.3.0*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

## Motivation

KAFKA-2798 (0.9.0.0) introduced the ability for each source connector and sink connector to inherit their client configurations from the worker properties. Within the worker properties, any configuration that has a prefix of "producer." or "consumer." are applied to all source connectors and sink connectors respectively. While the original proposal allowed overrides for source and sink connectors, it is still restrictive in terms of allowing different configurations for connectors. Typically, connect users would want to be able to do the following:-

- Use different principal for each connector so that they could control ACL at a fine-grained level
- Ability to optimize the producer and the consumer configurations for each connector so that connectors are set up for their performance characteristic

KIP-296: Connector level configurability for client configs aimed to solve this by allowing all configurations to be allowed to be overridden. But the KIP doesn't provide the ability for the connect operators to control what the connectors can override. Without this ability, there will be no clear line of separation between the connector and worker since the connector itself can now assume that the overrides would be available. But from an operational perspective, it will be good to have the following enforced

- Ability to control the config keys that can be overridden. For e.g. an administrator might never want the broker endpoint to be overridden
- Ability to control the allowed values for configs that are overridden. This helps with administrators defining the bounds of their clusters and manage multi-tenant cluster efficiently For e.g. the administrator might never want the `send.buffer.bytes` to above say 512 kb
- Ability to control the above based on connector types, client type (admin vs producer vs consumer), etc.

Based on the above context, the proposal is to allow an administrator to define/implement policy around what can be overridden very similar to the `Create TopicPolicy` which allows and controls the configurations to be specified at the topic level.

## Public Interfaces

On a high level, the proposal is to introduce a configurable policy similar to the CreateTopicPolicy available in the Core Kafka for Connector client Config Overrides. More specifically, we will introduce a new worker configuration that will allow an administrator to configure the policy for Connector Client config overrides.

#### New configuration

connector.client.config.override.policy - This will be an implementation of a new interface ConnectorClientConfigOverridePolicy that will be introduced in the connect API. The default value will be `None` which will not allow any overrides. Since the possibility of users already having config with the proposed prefixes is very slim, backward compatibility is generally not a problem. In the very rare case where users have these in their existing configs, they would have to just remove the configs to get it working again.

The overrides can be specified in the connector config by using the following prefixes

- `producer.override.` - Used for source connector's producer  & DLQ producer in the context of SinkConnector
- `consumer.override.` - Used for Sink Connector
- `admin.override.` - Used for DLQ topic create in Sink Connector ( The KIP will also allow DLQ settings to be specified in the worker using `admin` prefix to be consistent with producer & consumer)

The administrator could either specify the fully qualified class name of the ConnectorClientConfigOverridePolicy implementation or an alias (the alias is computed to be the prefix on the interface name `ConnectorClientConfigOverridePolicy` which is exactly how most of the existing connect plugins compute their alias).

The new interface will be treated as a new connect plugin and will be loaded via the plugin path mechanism. The plugins will be discovered via the Service loader mechanism similar to RestExtension and ConfigProvider.  The structure of the new interface and its request are described below:-

```
import org.apache.kafka.common.config.ConfigValue;

/**
 * <p>An interface for enforcing a policy on overriding of client configs via the connector configs.
 *
 * <p>Common use cases are ability to provide principal per connector, <code>sasl.jaas.config</code>
 * and/or enforcing that the producer/consumer configurations for optimizations are within acceptable ranges.
 */
public interface ConnectorClientConfigOverridePolicy extends Configurable, AutoCloseable {


    /**
     * Worker will invoke this while constructing the producer for the SourceConnectors,  DLQ for
SinkConnectors and the consumer for the
     * SinkConnectors to validate if all of the overridden client configurations are allowed per the
     * policy implementation. This would also be invoked during the validate of connector configs via the Rest
API.
     *
     * If there are any policy violations, the connector will not be started.
     *
     * @param connectorClientConfigRequest an instance of {@code ConnectorClientConfigRequest} that provides
the configs to overridden and
     *                                     its context; never {@code null}
     * @return List of Config, each Config should indicate if they are allowed via {@link
ConfigValue#errorMessages}
     */
    List<ConfigValue> validate(ConnectorClientConfigRequest connectorClientConfigRequest);
}
```

```
public class ConnectorClientConfigRequest {

    private Map<String, Object> clientProps;
    private ClientType  clientType;
    private String connectorName;
    private ConnectorType connectorType;
    private Class<? extends Connector> connectorClass;

    public ConnectorClientConfigRequest(
        String connectorName,
        ConnectorType connectorType,
        Class<? extends Connector> connectorClass,
        Map<String, Object> clientProps,
        ClientType clientType) {
        this.clientProps = clientProps;
        this.clientType = clientType;
        this.connectorName = connectorName;
        this.connectorType = connectorType;
        this.connectorClass = connectorClass;
    }

    /**
     * <pre>
     * Provides Config with prefix {@code producer.override.} for {@link ConnectorType#SOURCE}.
     * Provides Config with prefix {@code consumer.override.} for {@link ConnectorType#SINK}.
     * Provides Config with prefix {@code producer.override.} for {@link ConnectorType#SINK} for DLQ.
     * Provides Config with prefix {@code admin.override.} for {@link ConnectorType#SINK} for DLQ.
```

```
     * </pre>
     *
     * @return The client properties specified in the Connector Config with prefix {@code producer.override.} ,
     * {@code consumer.override.} and {@code admin.override.}. The configs returned don't include these
prefixes.
     */
    public Map<String, Object> clientProps() {
        return clientProps;
    }

    /**
     * <pre>
     * {@link ClientType#PRODUCER} for {@link ConnectorType#SOURCE}
     * {@link ClientType#CONSUMER} for {@link ConnectorType#SINK}
     * {@link ClientType#PRODUCER} for DLQ in {@link ConnectorType#SINK}
     * {@link ClientType#ADMIN} for DLQ  Topic Creation in {@link ConnectorType#SINK}
     * </pre>
     *
     * @return enumeration specifying the client type that is being overriden by the worker; never null.
     */
    public ClientType clientType() {
        return clientType;
    }

    /**
     * Name of the connector specified in the connector config.
     *
     * @return name of the connector; never null.
     */
    public String connectorName() {
        return connectorName;
    }

    /**
     * Type of the Connector.
     *
     * @return enumeration specifying the type of the connector {@link ConnectorType#SINK} or {@link
ConnectorType#SOURCE}.
     */
    public ConnectorType connectorType() {
        return connectorType;
    }

    /**
     * The class of the Connector.
     *
     * @return the class of the Connector being created; never null
     */
    public Class<? extends Connector> connectorClass() {
        return connectorClass;
    }

    public enum ClientType {
        PRODUCER, CONSUMER, ADMIN;
    }
}
```

The KIP introduces the following implementations of ConnectorClientConfigOverridePolicy that are outlined in the table below

| Class Name | Alias | Behavior |
|---|---|---|
| NoneConnectorClientConfigOverridePolicy | None | Disallows any configuration overrides. This will be the default policy. |
| PrincipalConnectorClientConfigOverridePolicy | Principal | Allows override of  "security.protocol", "sasl.jaas.config" and "sasl.mechanism" for the producer, consumer and admin prefixes.  Enables the ability to use different principal per connector. |
| AllConnectorClientConfigOverridePolicy | All | Allows override of all configurations for the producer, consumer and admin prefixes. |

Since the users can specify any of these policies, the connectors itself should not rely on these configurations to be available. The overrides are to be used purely from an operational perspective.

The policy itself will be enforced when a user attempts to either create the connector or validate the connector. When any of the ConfigValue has an error message

- During validate, the response will include error and the specific configurations that failed to meet the policy will also include the error message included in the response
- During create/update connector, the connector will fail to start

# Proposed Changes

As specified in the previous section, the design will include introducing a new worker configuration and an interface to define the override policy.

The worker would apply the policy during a create connector flow as follows. The configurations that are being overridden will be passed without the prefixes to the policy:-

- Constructing producer for WorkerSourceTask - invoke validate with all configs with "producer.override." prefix , ClientType=Producer, ConnectorType=Source & override if no policy violation
- Constructing admin client & producer for DeadLetterQueueReporter for the DLQ topic
  - invoke validate with all configs with "producer.override." prefix , ClientType=Producer, ConnectorType=Sink & override if no policy violation
  - invoke validate with all configs with "admin.override." prefix , ClientType=Admin, ConnectorType=Sink & override if no policy violation
- Constructing consumer for WorkerSinkTask - invoke validate with all configs with "consumer.override." prefix , ClientType=Consumer, ConnectorType=Sink & override if no policy violation

The herder(AbstractHerder) will apply the policy for all overrides as follows during the validate() flow. The configurations that are being overridden will be passed without the prefixes:-

- If its a source connector, apply the policy on each of the connector configurations with "producer." prefix and update the ConfigInfos result ( response of the validate API)
- If its a sink connector,
  - apply the policy on each of the connector configurations with "consumer." prefix and update the ConfigInfos result ( response of the validate API)
  - apply the policy on each of the connector configurations with "admin." prefix and update the ConfigInfos result when DLQ is enabled( response of the validate API)

# Compatibility, Deprecation, and Migration Plan

- The possibility of someone having connectors with the proposed prefixes is very slim and hence backward compatibility is not really a problem. In the rare case, if a user has configurations with these prefixes, they would either have to remove the config or alter the policy to get it working.

# Rejected Alternatives

- Override all configurations passed in the connector with the prefix 'producer.' or 'consumer.' - This doesn't provide control to the cluster administrator on what is an acceptable override.
- Override just the "sasl.jaas.config" from the connector - This is very restrictive in terms of what it can achieve
- Running multiple herders in the Connect cluster - This will reduce the ease of operation of a connect cluster since each connector would require a Herder to spun up within the cluster.