KIP 269 Substitution Within Configuration Values

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: *KAFKA-6664*

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

Motivation

The JAAS configuration for various SASL mechanisms would benefit from the ability to substitute values based on delimited text. For example, clients that connect via SASL/PLAIN (username/password) currently must specify the password directly in the configuration, like this:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
  username="kafkaclient1" \
  password="kafkaclient1-secret";
```

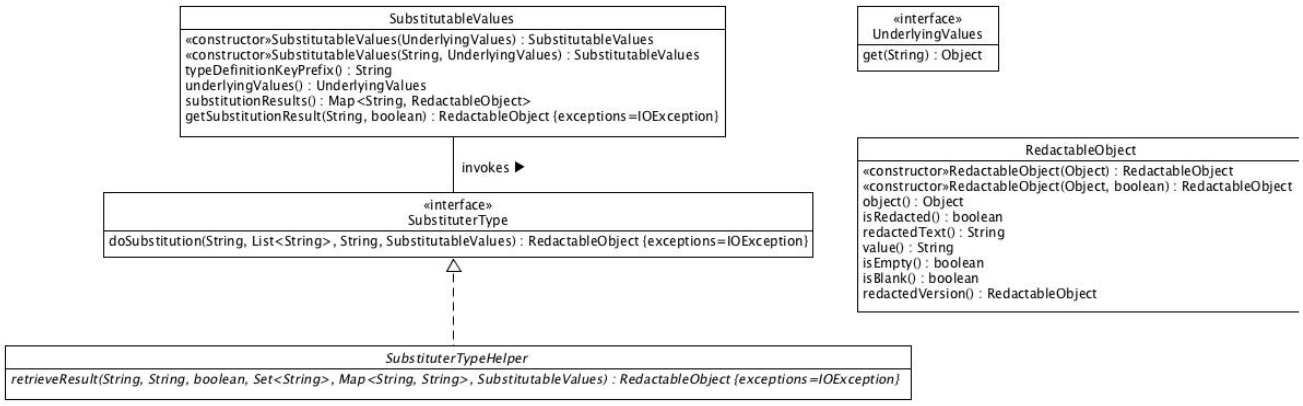
It would be a significant improvement both in security and flexibility if instead the password could be retrieved from elsewhere. For example:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
  username="kafkaclient1" \
  password="$[file=/path/to/secrets/kafkaclient1-secret]";
```

KIP 255 (OAuth Authentication via SASL/OAUTHBEARER) proposes adding a SASL mechanism that, due to the flexible nature of the OAuth 2 framework, will require significant configuration that cannot be predicted in advance; the ability to substitute values into the OAuth 2 configuration is a necessity. (In fact, adding the ability to substitute values into a JASS configuration originated with that KIP, but based on discussion the functionality was split out into this separate KIP when the general applicability was recognized.)

This KIP proposes adding support for substitution within client JAAS configuration values for PLAIN and SCRAM-related SASL mechanisms in a backwards-compatible manner and making the functionality available to other existing (or future) configuration contexts where it is deemed appropriate.

Public Interfaces



The public interface in terms of code is depicted in the above UML diagram. A set of 4 built-in substitution types will also be provided as described below. The *implementations* of the built-in substitution types are not part of the public code; rather, the way the substitutions are invoked within a JASS configuration will be the public interface to their functionality.

Here are the classes/interfaces as depicted in the above UML diagram. Note the following:

- The `UnderlyingValues` interface defines the map-like interface that the input to `SubstitutableValues` must implement. When a reference is made from one underlyng value to another – via the `defaultKey=<Key>` or `fromValueOfKey` modifiers as described later – the reference is resolved within the underlying values provided to the `SubstitutableValues`instance.
- Once an instance of `SubstitutableValues` retrieves an underlying value and its calculated value -- whether different from the raw underlying value due to a substitution or not -- is determined, the instance of `SubstitutableValues` will not retrieve the underlying value again; the calculated value will be used if it is referred to. This means if the underlying values are expected to change then to see those changes a new instance of `SubstitutableValues` must be allocated.
- A parsing error within recognized delimiters results in the delimiters and intervening text being passed through unchanged. This is to help prevent accidental substituion occuring in existing passwords (for example). See Compatibility below.

org.apache.kafka.common.security.substitutions.UnderlyingValues

```
package org.apache.kafka.common.security.substitutions;

/**
 * The map-like interface that the input to {@link SubstitutableValues} must
 * implement.
 */

public interface UnderlyingValues {
    /**
     * Return the value associated with the given key, if any, otherwise null
     *
     * @param key
     *            the mandatory key
     * @return the value associated with the given key, if any, otherwise null
     */
    Object get(String key);
}
```

org.apache.kafka.common.security.substitutions.SubstitutableValues

```
package org.apache.kafka.common.security.substitutions;

/**
 * Adds support for substitution within the values of an instance of
 * {@link UnderlyingValues}. Substitution is accomplished via delimited text
 * within a value of the following form:
 *
 * <pre>
 * &lt;OPENING_DELIMITER&gt;&lt;TYPE&gt;&lt;OPTIONAL_MODIFIERS&gt;=&lt;OPTIONAL_IDENTIFIER&gt;&lt;
CLOSING_DELIMITER&gt;
 * </pre>
 *
 * Where the above elements are defined as follows:
 *
 * <pre>
 * OPENING_DELIMITER: $[, $[[, $[[[, $[[[[, or $[[[[[
 * CLOSING_DELIMITER:  ],  ]],  ]]],  ]]]], or  ]]]]] (number of brackets must match)
 *
 * TYPE: everything up to (but not including) the first punctuation character
 *
 * OPTIONAL_MODIFIERS: the optional section immediately after the TYPE, starting with any
 *                     punctuation character except for the equal sign (=), and ending
 *                     with the same punctuation character followed immediately by an
 *                     equal sign. The same punctuation character delimits individual
 *                     modifiers, which come in two flavors: flags, which do not contain
 *                     an equal sign, and name=value arguments, which do.
 *
 * OPTIONAL_IDENTIFIER: the optional section immediately after any modifiers and the equal
 *                      sign (=).
 * </pre>
 *
 * For example:
 *
```

```
* <pre>
* $[envVar=THE_ENV_VAR]
* $[envVar/notBlank/redact/=THE_ENV_VAR]
* $[envVar/defaultValue = theDefaultValue/=THE_ENV_VAR]
* $[envVar/defaultKey = theDefaultKey/=THE_ENV_VAR]
* $[file|redact|notBlank|=/the/path/to/the/file]
* </pre>
*
* Once an underlying value is retrieved and its calculated value -- whether
* different from the raw underlying value due to a substitution or not -- is
* determined, this instance will not retrieve the underlying value again; the
* calculated value will be used if it is referred to. This means if the
* underlying values are expected to change then to see those changes a new
* instance of this class must be allocated.
* <p>
* Working left to right, once the delimiters are defined for a value (for
* example, {@code $[} and {@code ]}), only those delimiters are recognized for
* the rest of that value (and they are always recognized as meaning
* substitution for the rest of that value). A special "empty" substitution does
* nothing except, when it appears to the left of every other occurrence of
* matching delimiters, it serves to force the delimiter for that value to the
* one indicated. For example, to force the delimiter to {@code $[[} and
* {@code ]]} (and prevent {@code $[} and {@code ]} from causing substitution)
* for a value:
*
* <pre>
* someKey = "$[[]]These $[ and ] delimiters do not cause substitution"
* </pre>
*
* The following built-in substitution types are supported, though it is
* straightforward to add others (see below):
*
* <ul>
* <li>{@code envVar}: substitute an environment variable, typically indicated
* by the identifier</li>
* <li>{@code sysProp}: substitute a system property, typically indicated by the
* identifier</li>
* <li>{@code file}: substitute the contents of a file, typically indicated by
* the identifier</li>
* <li>{@code keyValue}: substitute the contents of another key's value,
* typically indicated by the identifier (and that key's value has substitution
* performed on it if necessary)</li>
* </ul>
*
* The built-in substitution types support the following flags, which are
* trimmed and may be redundantly specified:
* <ul>
* <li>{@code redact}: prevent values from being logged</li>
* <li>{@code notEmpty}: the value must not be empty</li>
* <li>{@code notBlank}: the value must not be blank (i.e. consisting only of
* whitespace); implies {@code notEmpty}.</li>
* <li>{@code fromValueOfKey}: provides a level of indirection so that the
* identifier, instead of being used directly (i.e. read the indicated file, or
* the indicated environment variable), identifies another key whose value is
* used as the identifier instead. This allows, for example, the filename,
* system property name, etc. to potentially be generated from multiple
* substitutions concatenated together.</li>
* </ul>
* The built-in substitution types support the following arguments, whose names
* are trimmed but whose values are not; it is an error to specify the same
* named argument multiple times (even if the values are identical):
* <ul>
* <li>{@code defaultValue=<value>}: substitute the given literal value if the
* substitution cannot otherwise be made (either because the identifier
* indicates something that does not exist or the determined value was
* disallowed because it was empty or blank). The substituted default value must
* satisfy any {@code notBlank} or {@code notEmpty} modifiers that act as
* constraints, otherwise it is an error.</li>
* <li>{@code defaultKey=<key>}: substitute the value associated with the
* indicated key if the substitution cannot otherwise be made (either because
* the identifier indicates something that does not exist or the determined
```

```
 * value was disallowed because it was empty or blank). The value that is
 * ultimately substituted must satisfy any {@code notBlank} or {@code notEmpty}
 * modifiers that act as constraints, otherwise it is an error.</li>
 * </ul>
 *
 * To add new substitutions beyond the built-in ones mentioned above simply
 * define a key/value pair of the following form:
 *
 * <pre>
 * [optionalTypeDefinitionKeyPrefix]&lt;type&gt;SubstituterType = "fully.qualified.class.name"
 * </pre>
 *
 * For example:
 *
 * <pre>
 * fooSubstituterType = "org.example.FooSubstituterType"
 * </pre>
 *
 * The indicated class must implement {@link SubstituterType}, and you can
 * invoke the substitution in a value like this:
 *
 * <pre>
 * $[foo/optional/modifiers/=optionalValue]
 * </pre>
 *
 * The type definition prefix is defined at construction time and may be empty.
 *
 * A parsing error within recognized delimiters results in the delimiters and
 * the intervening text that could not be parsed being left alone. For example,
 * the following text would be passed through unchanged because the delimited
 * text cannot be parsed as a valid substitution request:
 * {@code qw$[asd_4Q!]uH6}.
 *
 * @see SubstituterType
 * @see SubstituterTypeHelper
 */
public class SubstitutableValues {
    /**
     * Constructor where the type definition key prefix is empty
     *
     * @param underlyingMap
     *            the mandatory underlying map. It is not copied, so it should be
     *            immutable. Results are unspecified if it is mutated in any manner.
     */
    public SubstitutableValues(UnderlyingValues underlyingValues) {
        this("", underlyingValues);
    }

    /**
     * Constructor with the given type definition key prefix
     *
     * @param typeDefinitionKeyPrefix
     *            the mandatory (but possibly empty) type definition key prefix
     * @param underlyingMap
     *            the mandatory underlying map. It is not copied, so it should be
     *            immutable. Results are unspecified if it is mutated in any manner.
     */
    public SubstitutableValues(String typeDefinitionKeyPrefix, UnderlyingValues underlyingValues) {
        // etc...
    }
    /**
     * Return the always non-null (but possibly empty) type definition key prefix
     *
     * @return the always non-null (but possibly empty) type definition key prefix
     */
    public String typeDefinitionKeyPrefix() {
        return typeDefinitionKeyPrefix;
    }


    /**
```

```
     * Return the underlying values provided during construction
     *
     * @return the underlying values provided during construction
     */
    public UnderlyingValues underlyingValues() {
        return underlyingValues;
    }
    /**
     * Return an unmodifiable map identifying which keys have been processed for
     * substitution and the corresponding result (if any). A key is guaranteed to
     * have been processed for substitution and its name will appear as a key in the
     * returned map only after {@link #getSubstitutionResult(String)} has been
     * invoked for that key either directly or indirectly because some other key's
     * substitution result depends on the substitution result of the key.
     *
     * @return an unmodifiable map identifying which keys have been processed for
     *         substitution and the corresponding result (if any)
     */
    public Map<String, RedactableObject> substitutionResults() {
        // etc...
    }

    /**
     * Perform substitution if necessary and return the resulting value for the
     * given key
     *
     * @param key
     *            the mandatory requested key
     * @param requiredToExist
     *            if true then the requested key is required to exist
     * @return the given key's substitution result, after any required substitution
     *         is applied, or null if the key does not exist and it was not required
     *         to exist
     * @throws IOException
     *             if a required substitution cannot be performed, including if the
     *             given (or any other) required key does not exist
     */
    public RedactableObject getSubstitutionResult(String key, boolean requiredToExist) throws IOException {
        // etc...
    }

    // etc...
}
```

org.apache.kafka.common.security.substitutions.SubstituterType

```
package org.apache.kafka.common.security.substitutions;

/**
 * The single-method interface that pluggable substituter types must implement.
 */
public interface SubstituterType {
    /**
     * Perform the substitution of the given type using the given modifiers and
     * value on the given options
     *
     * @param type
     *            the (always non-null) type of substitution to perform
     * @param modifiers
     *            the (always non-null but potentially empty) modifiers to apply, if
     *            any. They are presented exactly as they appear in the
     *            configuration, with no whitespace trimming applied.
     * @param identifier
     *            the always non-null (but potentially empty) identifier, which is
     *            interpreted in the context of the substitution of the indicated
     *            type. For example, it may indicate an environment variable name, a
     *            filename, etc.
     * @param substitutableValues
     *            the values and their current substitution state
     * @return the (always non-null) result of performing the substitution
     * @throws IOException
     *             if the substitution cannot be performed
     */
    RedactableObject doSubstitution(String type, List<String> modifiers, String identifier,
            SubstitutableValues substitutableValues) throws IOException;
}
```

org.apache.kafka.common.security.substitutions.SubstituterTypeHelper

```
package org.apache.kafka.common.security.substitutions;

/**
 * A template {@code SubstituterType} that handles the following modifiers:
 * <ul>
 * <li>{@code redact} -- when enabled, results are stored such that they are
 * prevented from being logged</li>
 * <li>{@code notBlank} -- when enabled, blank (only whitespace) or non-existent
 * results are replaced by default values. Implies {@code notEmpty}.</li>
 * <li>{@code notEmpty} -- when enabled, either explicitly or via
 * {@code notBlank}, empty ({@code ""}) or non-existent results are replaced by
 * default values.</li>
 * <li>{@code fromValueOfKey} -- provides a level of indirection so that the
 * identifier, instead of always being literally specified (i.e. read this
 * particular file, or this particular environment variable), can be determined
 * via some other key's value. This allows, for example, the filename, system
 * property name, etc. to potentially be generated from multiple substitutions
 * concatenated together.</li>
 * <li>{@code defaultValue=<value>} -- when enabled, the provided literal value
 * is used as a default value in case the result either does not exist or is
 * disallowed via {@code notBlank} or {@code notEmpty}</li>
 * <li>{@code defaultKey=<key>} -- when enabled, the indicated key is evaluated
 * as a default value in case the result either does not exist or is disallowed
 * via {@code notBlank} or {@code notEmpty}</li>
 * </ul>
 *
 * Flags (modifiers without an equal sign) are trimmed, so "{@code redact}" and
 * "{@code  redact }" are recognized as being the same. Arguments (modifiers
 * with an equal sign) have their name trimmed but not their value, so
 * "{@code name=value}" and "{@code  name = value }" are both recognized as
 * setting the {@code name} argument (though their values do not match due to
 * whitespace differences).
```

```
 * <p>
 * It is an error to set the same named argument multiple times (even if the
 * values are the same). Redundantly specifying the same flag is acceptable.
 * <p>
 * Flags and arguments are presented to the substitution type's implementation
 * via the
 * {@link #retrieveResult(String, String, boolean, Set, Map, SubstitutableValues)}
 * method.
 * <p>
 * Implementations of the {@link SubstituterType} interface that wish to
 * leverage the help provided by this class can either extend this class
 * directly or delegate to an anonymous class that extends this one.
 */
public abstract class SubstituterTypeHelper implements SubstituterType {
    /**
     * Retrieve the substitution result associated with the given identifier, if
     * any, otherwise null
     *
     * @param type
     *            the (always non-null/non-blank) type of substitution being
     *            performed
     * @param identifier
     *            the required (though potentially empty) identifier as interpreted
     *            by the substitution implementation for the given type. For
     *            example, it may be a filename, system property name, environment
     *            variable name, etc.
     * @param redact
     *            if the result must be redacted regardless of any information to
     *            the contrary
     * @param additionalFlags
     *            the flags specified, if any, beyond the standard {@code redact},
     *            {@code notBlank}, {@code notEmpty}, and {@code fromValueOfKey}
     *            flags
     * @param additionalArgs
     *            the arguments specified, if any, beyond the standard
     *            {@code defaultValue} and {@code defaultKey} arguments
     * @param substitutableValues
     *            the key/value mappings and their current substitution state
     * @return the substitution result associated with the given identifier, if any,
     *         otherwise null
     * @throws IOException
     *             if the request cannot be performed such that the use of a default
     *             value would be inappropriate
     */
    public abstract RedactableObject retrieveResult(String type, String identifier, boolean redact,
            Set<String> additionalFlags, Map<String, String> additionalArgs, SubstitutableValues
substitutableValues)
            throws IOException;

    @Override
    public RedactableObject doSubstitution(String type, List<String> modifiers, String identifier,
            SubstitutableValues substitutableValues) throws IOException {
        // etc...
    }

    // etc...
}
```

org.apache.kafka.common.security.substitutions.RedactableObject

```
package org.apache.kafka.common.security.substitutions;

/**
 * An object whose text value can be redacted
 */
public class RedactableObject {
    static final String REDACTED = "[redacted]";
    private final Object object;
    private final boolean redacted;
```

```
/**
 * Constructor for an instance that will be redacted only if the given object is
 * of type {@link Password}.
 *
 * @param object
 *            the mandatory object
 */
public RedactableObject(Object object) {
    this(Objects.requireNonNull(object), object instanceof Password);
}

/**
 * Constructor
 *
 * @param object
 *            the mandatory object
 * @param redact
 *            when true the object's value will be redacted in
 *            {@link #redactedText()}
 */
public RedactableObject(Object object, boolean redact) {
    this.object = Objects.requireNonNull(object);
    this.redacted = redact;
}

/**
 * Return the (always non-null) underlying object provided during instance
 * construction
 *
 * @return the (always non-null) underlying object provided during instance
 *         construction
 */
public Object object() {
    return object;
}

/**
 * Return true if this instance contains information that will be redacted when
 * {@link #redactedText()} is invoked, otherwise false
 *
 * @return true if this instance contains information that will be redacted when
 *         {@link #redactedText()} is invoked, otherwise false
 */
public boolean isRedacted() {
    return redacted;
}

/**
 * Return the redacted text for this instance, if redaction is required,
 * otherwise return the {@link #value()}
 *
 * @return the redacted text for this instance, if redaction is required,
 *         otherwise return the {@link #value()}
 */
public String redactedText() {
    return redacted ? REDACTED : value();
}

/**
 * Return the {@code String} value of this instance, including information that
 * would otherwise be redacted
 *
 * @return the {@code String} value of this instance, including information that
 *         would otherwise be redacted
 */
public String value() {
    if (object instanceof String)
        return (String) object;
    if (object instanceof Password)
        return ((Password) object).value();
```

```
        return object.toString();
    }

    /**
     * Return true if this result is considered to be empty, otherwise false
     *
     * @return true if this result is considered to be empty, otherwise false
     */
    public boolean isEmpty() {
        return value().isEmpty();
    }

    /**
     * Return true if this result is considered to be blank (containing at most just
     * whitespace), otherwise false
     *
     * @return true if this result is considered to be blank (containing at most
     *         just whitespace), otherwise false
     */
    public boolean isBlank() {
        return value().trim().isEmpty();
    }

    /**
     * Return this instance if it is redacted according to {@link #isRedacted()},
     * otherwise return a new, redacted instance with the same underlying object
     *
     * @return this instance if it is redacted according to {@link #isRedacted()},
     *         otherwise return a new, redacted instance with the same underlying
     *         object
     */
    public RedactableObject redactedVersion() {
        return redacted ? this : new RedactableObject(object, true);
    }

    @Override
    public String toString() {
        // be sure to redact information as required
        return redactedText();
    }

    // etc...
}
```

Here is more detail about the built-in substitution types, how to invoke them from within a JAAS configuration, and how to add new substitution types.

As an example to help illustrate the sytnax, the following would support substitution of the contents of a file (which is a common way to store secrets, especially within containers):

```
thePassword="$[file|redact|notBlank|defaultKey=fileDefault|=/path/to/secrets/the_secret]"
```

There are several features here that deserve comment:

- The "`$[`" and "`]`" **delimiters** are the signal to perform a substitution (we can't use "`${`" and "`}`" because that is already defined by the JAAS Configuration spec to mean system property substitution). Note that we will **not** allow substitution within a substitution, and in fact it is not needed as described below. We will also support "`$[[`" and "`]]`" as delimiters (all the way up to 5 brackets, actually) to allow "`$[`" and "`]`" to appear in text without causing substitution.
- Immediately inside the opening delimiter is the **type** of substitution followed by any optional **modifiers** we wish to apply. In the above, we identify this as a file substitution and we indicate three modifiers: the resulting value should never be logged (i.e. store it such that its value will be redacted when logged); the contents of the file must not be blank (meaning it must not be empty or only contain whitespace); if the file does not exist or its contents are blank then use the value of the "`fileDefault`" key in the configuration (which could itself have substitutions). It is an error if any constraints implied by the modifiers are violated. Any punctuation character except the equal sign (=) can be used to delimit the modifiers.
- Immediately after the type of substitution and any optional modifiers is an equal sign ("=") followed by the **identifier** (which in the case of the "`file`" type is interpreted as the filename to read); then ultimately the closing delimiter appears.

This scheme is flexible and powerful; it handles most cases, but it remains relatively easy to create and read. Importantly, the types of replacements can be expanded in the future without breaking compatibility.

The initial set of supported substitution types and their supported modifiers are as follows:

Type	Description	Specifiable Modifiers	Notes

file	File content substitution	`notBlank`, `notEmpty`, `redact`, `fromValueOfKey`
		
		`defaultValue=<value>`, `defaultKey=<Key>`
			The identifier typically specifies the file to read. It is an error if the file does not exist or is not readable unless `defaultValue` or `defaultKey` is specified. If a `defaultValue` is specified then the literal default value specified will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If a `defaultKey` is specified then the value defined by the specified key will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If the default key's value depends on substitutions that were marked `redact` then `redact` is implied. The `fromValueOfKey` modifier indicates that the identifier, instead of being the file to read, instead identifies the key whose value in turn is to be taken as the filename. This provides the ability to generate filenames from multiple substitutions as opposed to being forced to literally specify it. It is an error to try to read a file that is larger than 1 MB in size.

envVar	Environment variable substitution	same as above	The identifier typically specifies the environment variable to read. It is an error if the environment variable does not exist unless `defaultValue` or `defaultKey` is specified. If a `defaultValue` is specified then the literal default value specified will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If a `defaultKey` is specified then the value defined by the specified key will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If the default key's value depends on substitutions that were marked `redact` then `redact` is implied. The `fromValueOfKey` modifier indicates that the identifier, instead of being the environment variable to read, instead identifies the key whose value in turn is to be taken as the environment variable name. This provides the ability to generate environment variable names from multiple substitutions as opposed to being forced to literally specify it.
keyValue	Substitution of another key's value	same as above	The identifier typically specifies the key whose value is to be read. It is an error if the option does not exist unless `defaultValue` or `defaultKey` is specified. If a `defaultValue` is specified then the literal default value specified will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If a `defaultKey` is specified then the value defined by the specified key will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If the default key's value depends on substitutions that were marked `redact` then `redact` is implied. The `fromValueOfKey` modifier indicates that the identifier, instead of being the key to read, instead identifies the key whose value in turn is to be taken as the key. This provides the ability to generate a key from multiple substitutions as opposed to being forced to literally specify it.
sysProp	System property substitution	same as above	The identifier typically specifies the system property to read. It is an error if the system property does not exist unless `defaultValue` or `defaultKey` is specified. If a `defaultValue` is specified then the literal default value specified will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If a `defaultKey` is specified then the value defined by the specified key will be used and checked against any `notBlank` or `notEmpty` constraints that exist if those constraints are violated by the previously-determined value. If the default key's value depends on substitutions that were marked `redact` then `redact` is implied. The `fromValueOfKey` modifier indicates that the identifier, instead of being the system property to read, instead identifies the key whose value in turn is to be taken as the system property name. This provides the ability to generate system property names from multiple substitutions as opposed to being forced to literally specify it.

To add new substitutions simply define a key in the configuration of the following form:

```
[optionalTypeDefinitionKeyPrefix]<type>SubstituterType = "fully.qualified.class.name"
```

For example:

```
fooSubstituterType = "org.example.FooSubstituterType"
```

The indicated class must implement the `org.apache.kafka.common.security.substitutions.SubstituterType` interface. It is recommended (though not required) that new substitution types leverage the `org.apache.kafka.common.security.substitutions.SubstituterTypeHelper` class.

Invoke the substitution with text in a key's value like this:

```
$[foo/optional/modifiers/=optionalIdentifier]
```

Proposed Changes

This KIP proposes adding the above classes to support substitution into configuration values where it is deemed appropriate. Specifically, this KIP proposes adding support for substitution within the configuration read by the following classes to allow clients leveraging the associated SASL mechanisms to retrieve their username and password from elsewhere if they so choose:

- `org.apache.kafka.common.security.plain.PlainLoginModule`
- `org.apache.kafka.common.security.scram.ScramLoginModule`

Note that it would likely be possible to support substitution into configuration values in contexts other than client JAAS configurations (for example, server JAAS configurations, or perhaps even the cluster configuration), but this KIP does **not** propose any of these possibilities. If any such changes are desired then they should be proposed via separate KIPs for discussion.

Compatibility, Deprecation, and Migration Plan

There is a possibility that existing usernames or (more likely) passwords in existing client JAAS configurations could contain the "`$[`" and "`]`" delimiters. This would cause a substitution to be attempted, which of course would fail and potentially raise an exception. This risk is low, but it nonetheless does need to be mitigated; therefore any already-existing login modules where substitution support is to be added (namely, the ones mentioned above) will **only enable substitution if a key/value pair is explicitly added to the JAAS configuration as follows**:

```
enableSubstitution="true"
```

Existing behavior will remain unchanged in the absence of this explicit opt-in key/value pair. Even with this opt-in, though, we still do not want unintended substitutions to occur, so if delimiters are recognized but a parsing error occurs (e.g. a value such as `qw$[asd_4Q!]uH6` would cause a substitution to be attempted but does not ultimately meet the required syntax) the delimiters and intervening text will be passed unchanged.

Rejected Alternatives

This KIP does not define `Callback` or `CallbackHandler` implementations because configuration values are typically retrieved without using them (this is the case with `PlainLoginModule` and `ScramLoginModule`).