KIP-504 - Add new Java Authorizer Interface

Status

Current state: *"Accepted"*

Discussion thread: *here*

JIRA: **KAFKA-8865** - Getting issue details... STATUS

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

Motivation

Kafka supports pluggable authorization using the Scala trait `kafka.security.auth.Authorizer`. KIP-50 was accepted to replace this with a Java interface and Java ACL classes in the package `org.apache.kafka.common` in the `'clients'` module. But this was never merged. Since KIP-50 was accepted, we have added new broker-side pluggable interfaces as Java interfaces in `'clients'` in the package `` `org.apache.kafka.server` ``. These Java interfaces provide a better compatibility story than Scala traits, allowing us to evolve the API.

This KIP is a replacement for KIP-50 to introduce a new Java interface for authorization. KIP-50 proposed to break compatibility because it was in the early days of adoption of authorization in Kafka. Since Kafka authorization has been widely adopted since then, we propose to deprecate, but continue to support the old interface to avoid breaking existing deployments during upgrade. We will also address the known limitations of the existing interface.

Goals

1. Define a new Java interface for authorizer in '`clients`' module in the package '`org.apache.kafka.server`' similar to other server-side pluggable classes.
2. KIP-4 has added ACL-related classes in the package `org.apache.kafka.common` (e.g. `ResourcePattern` and `AccessControlEntry`) to support ACL management using `AdminClient`. We will attempt to reuse these classes wherever possible.
3. Deprecate but retain existing Scala authorizer API for backward compatibility to ensure that existing custom authorizers can be used with new brokers.
4. Provide context about the request to authorizers to enable context-specific logic based on security protocol or listener to be applied to authorization.
5. Provide additional context about the request including `ApiKey` and correlation id from the request header since these are useful for matching debug-level authorization logs with corresponding request logs.
6. For ACL updates, provide request context including principal requesting the update and the listener on which request arrived to enable additional validation.
7. Return individual responses for each access control entry update when multiple entries of a resource are updated. At the moment, we update the ZooKeeper node for a resource pattern multiple times when a request adds or removes multiple entries for a resource in a single update request. Since it is a common usage pattern to add or remove multiple access control entries while updating ACLs for a resource, batched updates will be supported to enable a single atomic update for each resource pattern.
8. Provide authorization usage flag to authorizers to enable authorization logs to indicate attempts to access unauthorized resources. Kafka brokers log denied operations at `INFO` level and allowed operations at `DEBUG` level with the expectation that denied operations are rare and indicate erroneous or malicious use of the system. But we currently have several uses of `Authorizer#authorize` for filtering accessible resources or operations, for example for regex subscription. These fill up authorization logs with denied log entries, making these logs unusable for determining actual attempts to access resources by users who don't have appropriate permissions. Audit flag will enable the authorizer to determine the severity of denied access.

9. For authorizers that don't store metadata in ZooKeeper, ensure that authorizer metadata for each listener is available before starting up the listener. This enables different authorization metadata stores for different listeners.
10. Add a new out-of-the-box authorizer class that implements the new authorizer interface, making use of the features supported by the new API.
11. Retain existing audit log entry format in `SimpleAclAuthorizer` to ensure that tools that parse these logs continue to work.
12. Enable `Authorizer` implementations to make use of additional Kafka interfaces similar to other pluggable callbacks. Authorizers can implement `org.apache.kafka.common.Reconfigurable` to support dynamic reconfiguration without restarting the broker. Authorizers will also be provided cluster id which may be included in logs or used to support centralized ACL storage.
13. Enable asynchronous ACL updates to avoid blocking broker request threads when ACLs are updated in a remote store (e.g. a database).

Public Interfaces

Authorizer Configuration

The existing configuration option `authorizer.class.name` will be extended to support broker authorizers using the new Java interface `org.apache.kafka.server.authorizer.Authorizer` . The config will continue to support authorizers using the existing Scala trait `kafka.security.auth.Authorizer`

Authorizer API

The new Java `Authorizer` interface and its supporting classes are shown below:

Java Authorizer Interface

```
package org.apache.kafka.server.authorizer;

import java.io.Closeable;
import java.util.List;
import java.util.Map;
import java.util.concurrent.CompletionStage;
import org.apache.kafka.common.Configurable;
import org.apache.kafka.common.Endpoint;
import org.apache.kafka.common.acl.AclBinding;
import org.apache.kafka.common.acl.AclBindingFilter;
import org.apache.kafka.common.annotation.InterfaceStability;

/**
 *
 * Pluggable authorizer interface for Kafka brokers.
 *
 * Startup sequence in brokers:
 * <ol>
 *   <li>Broker creates authorizer instance if configured in `authorizer.class.name`.</li>
 *   <li>Broker configures and starts authorizer instance. Authorizer implementation starts loading its
metadata.</li>
 *   <li>Broker starts SocketServer to accept connections and process requests.</li>
 *   <li>For each listener, SocketServer waits for authorization metadata to be available in the
 *       authorizer before accepting connections. The future returned by {@link #start(AuthorizerServerInfo)}
 *       must return only when authorizer is ready to authorize requests on the listener.</li>
 *   <li>Broker accepts connections. For each connection, broker performs authentication and then accepts Kafka
requests.
 *       For each request, broker invokes {@link #authorize(AuthorizableRequestContext, List)} to authorize
 *       actions performed by the request.</li>
 * </ol>
 *
 * Authorizer implementation class may optionally implement @{@link org.apache.kafka.common.Reconfigurable}
 * to enable dynamic reconfiguration without restarting the broker.
 * <p>
 * <b>Threading model:</b>
 * <ul>
 *   <li>All authorizer operations including authorization and ACL updates must be thread-safe.</li>
 *   <li>ACL update methods are asynchronous. Implementations with low update latency may return a
 *       completed future using {@link java.util.concurrent.CompletableFuture#completedFuture(Object)}.
 *       This ensures that the request will be handled synchronously by the caller without using a
 *       purgatory to wait for the result. If ACL updates require remote communication which may block,
 *       return a future that is completed asynchronously when the remote operation completes. This enables
 *       the caller to process other requests on the request threads without blocking.</li>
```

```
 *   <li>Any threads or thread pools used for processing remote operations asynchronously can be started during
 *       {@link #start(AuthorizerServerInfo)}. These threads must be shutdown during {@link Authorizer#close()}.
</li>
 * </ul>
 * </p>
 */
@InterfaceStability.Evolving
public interface Authorizer extends Configurable, Closeable {

    /**
     * Starts loading authorization metadata and returns futures that can be used to wait until
     * metadata for authorizing requests on each listener is available. Each listener will be
     * started only after its metadata is available and authorizer is ready to start authorizing
     * requests on that listener.
     *
     * @param serverInfo Metadata for the broker including broker id and listener endpoints
     * @return CompletionStage for each endpoint that completes when authorizer is ready to
     *         start authorizing requests on that listener.
     */
    Map<Endpoint, ? extends CompletionStage<Void>> start(AuthorizerServerInfo serverInfo);

    /**
     * Authorizes the specified action. Additional metadata for the action is specified
     * in `requestContext`.
     * <p>
     * This is a synchronous API designed for use with locally cached ACLs. Since this method is invoked on the
     * request thread while processing each request, implementations of this method should avoid time-consuming
     * remote communication that may block request threads.
     *
     * @param requestContext Request context including request type, security protocol and listener name
     * @param actions Actions being authorized including resource and operation for each action
     * @return List of authorization results for each action in the same order as the provided actions
     */
    List<AuthorizationResult> authorize(AuthorizableRequestContext requestContext, List<Action> actions);

    /**
     * Creates new ACL bindings.
     * <p>
     * This is an asynchronous API that enables the caller to avoid blocking during the update. Implementations
of this
     * API can return completed futures using {@link java.util.concurrent.CompletableFuture#completedFuture
(Object)}
     * to process the update synchronously on the request thread.
     *
     * @param requestContext Request context if the ACL is being created by a broker to handle
     *        a client request to create ACLs. This may be null if ACLs are created directly in ZooKeeper
     *        using AclCommand.
     * @param aclBindings ACL bindings to create
     *
     * @return Create result for each ACL binding in the same order as in the input list. Each result
     *         is returned as a CompletionStage that completes when the result is available.
     */
    List<? extends CompletionStage<AclCreateResult>> createAcls(AuthorizableRequestContext requestContext,
List<AclBinding> aclBindings);

    /**
     * Deletes all ACL bindings that match the provided filters.
     * <p>
     * This is an asynchronous API that enables the caller to avoid blocking during the update. Implementations
of this
     * API can return completed futures using {@link java.util.concurrent.CompletableFuture#completedFuture
(Object)}
     * to process the update synchronously on the request thread.
     *
     * @param requestContext Request context if the ACL is being deleted by a broker to handle
     *        a client request to delete ACLs. This may be null if ACLs are deleted directly in ZooKeeper
     *        using AclCommand.
     * @param aclBindingFilters Filters to match ACL bindings that are to be deleted
     *
     * @return Delete result for each filter in the same order as in the input list.
     *         Each result indicates which ACL bindings were actually deleted as well as any
```

```
     *         bindings that matched but could not be deleted. Each result is returned as a
     *         CompletionStage that completes when the result is available.
     */
    List<? extends CompletionStage<AclDeleteResult>> deleteAcls(AuthorizableRequestContext requestContext,
List<AclBindingFilter> aclBindingFilters);

    /**
     * Returns ACL bindings which match the provided filter.
     * <p>
     * This is a synchronous API designed for use with locally cached ACLs. This method is invoked on the
request
     * thread while processing DescribeAcls requests and should avoid time-consuming remote communication that
may
     * block request threads.
     *
     * @return Iterator for ACL bindings, which may be populated lazily.
     */
    Iterable<AclBinding> acls(AclBindingFilter filter);
}
```

Request context will be provided to authorizers using a new interface `AuthorizableRequestContext`. The existing class `org.apache.kafka.common.requests.RequestContext` will implement this interface. New methods may be added to this interface in future, so mock implementations using this interface should adapt to these changes.

Request Context

```
package org.apache.kafka.server.authorizer;

import java.net.InetAddress;
import org.apache.kafka.common.annotation.InterfaceStability;
import org.apache.kafka.common.security.auth.KafkaPrincipal;
import org.apache.kafka.common.security.auth.SecurityProtocol;

/**
 * Request context interface that provides data from request header as well as connection
 * and authentication information to plugins.
 */
@InterfaceStability.Evolving
public interface AuthorizableRequestContext {

    /**
     * Returns name of listener on which request was received.
     */
    String listenerName();

    /**
     * Returns the security protocol for the listener on which request was received.
     */
    SecurityProtocol securityProtocol();

    /**
     * Returns authenticated principal for the connection on which request was received.
     */
    KafkaPrincipal principal();

    /**
     * Returns client IP address from which request was sent.
     */
    InetAddress clientAddress();

    /**
     * 16-bit API key of the request from the request header. See
     * https://kafka.apache.org/protocol#protocol_api_keys for request types.
     */
    int requestType();

    /**
     * Returns the request version from the request header.
     */
    int requestVersion();

    /**
     * Returns the client id from the request header.
     */
    String clientId();

    /**
     * Returns the correlation id from the request header.
     */
    int correlationId();
}
```

`AuthorizerServerInfo` provides runtime broker configuration to authorization plugins including broker id, cluster id and endpoint information. New methods may be added to this interface in future, so mock implementations using this interface should adapt to these changes.

Broker Runtime Config

```
package org.apache.kafka.server.authorizer;

import java.util.Collection;
import org.apache.kafka.common.ClusterResource;
import org.apache.kafka.common.Endpoint;
import org.apache.kafka.common.annotation.InterfaceStability;

/**
 * Runtime broker configuration metadata provided to authorizers during start up.
 */
@InterfaceStability.Evolving
public interface AuthorizerServerInfo {

    /**
     * Returns cluster metadata for the broker running this authorizer including cluster id.
     */
    ClusterResource clusterResource();

    /**
     * Returns broker id. This may be a generated broker id if `broker.id` was not configured.
     */
    int brokerId();

    /**
     * Returns endpoints for all listeners including the advertised host and port to which
     * the listener is bound.
     */
    Collection<Endpoint> endpoints();

    /**
     * Returns the inter-broker endpoint. This is one of the endpoints returned by {@link #endpoints()}.
     */
    Endpoint interBrokerEndpoint();
}
```

`Endpoint` is added as a common class so that it may be reused in several places in the code where we use this abstraction.

Endpoint

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

import java.util.Objects;
import java.util.Optional;

import org.apache.kafka.common.annotation.InterfaceStability;
import org.apache.kafka.common.security.auth.SecurityProtocol;

/**
 * Represents a broker endpoint.
 */

@InterfaceStability.Evolving
public class Endpoint {

    private final String listenerName;
    private final SecurityProtocol securityProtocol;
    private final String host;
    private final int port;

    public Endpoint(String listenerName, SecurityProtocol securityProtocol, String host, int port) {
        this.listenerName = listenerName;
        this.securityProtocol = securityProtocol;
```

```
        this.host = host;
        this.port = port;
    }

    /**
     * Returns the listener name of this endpoint. This is non-empty for endpoints provided
     * to broker plugins, but may be empty when used in clients.
     */
    public Optional<String> listenerName() {
        return Optional.ofNullable(listenerName);
    }

    /**
     * Returns the security protocol of this endpoint.
     */
    public SecurityProtocol securityProtocol() {
        return securityProtocol;
    }

    /**
     * Returns advertised host name of this endpoint.
     */
    public String host() {
        return host;
    }

    /**
     * Returns the port to which the listener is bound.
     */
    public int port() {
        return port;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Endpoint)) {
            return false;
        }

        Endpoint that = (Endpoint) o;
        return Objects.equals(this.listenerName, that.listenerName) &&
            Objects.equals(this.securityProtocol, that.securityProtocol) &&
            Objects.equals(this.host, that.host) &&
            this.port == that.port;

    }

    @Override
    public int hashCode() {
        return Objects.hash(listenerName, securityProtocol, host, port);
    }

    @Override
    public String toString() {
        return "Endpoint(" +
            "listenerName='" + listenerName + '\'' +
            ", securityProtocol=" + securityProtocol +
            ", host='" + host + '\'' +
            ", port=" + port +
            ')';
    }
}
```

`Action` provides details of the action being authorized including resource and operation. Additional context including audit flag indicating authorization usage are also included, enabling access violation to be distinguished from resource filtering or optional ACLs.

Authorizable Action

```
package org.apache.kafka.server.authorizer;

import java.util.Objects;
import org.apache.kafka.common.acl.AclOperation;
import org.apache.kafka.common.annotation.InterfaceStability;
import org.apache.kafka.common.resource.PatternType;
import org.apache.kafka.common.resource.ResourcePattern;
import org.apache.kafka.common.resource.ResourceType;

@InterfaceStability.Evolving
public class Action {

    private final ResourcePattern resourcePattern;
    private final AclOperation operation;
    private final int resourceReferenceCount;
    private final boolean logIfAllowed;
    private final boolean logIfDenied;

    public Action(AclOperation operation,
                  ResourcePattern resourcePattern,
                  int resourceReferenceCount,
                  boolean logIfAllowed,
                  boolean logIfDenied) {
        this.operation = operation;
        this.resourcePattern = resourcePattern;
        this.logIfAllowed = logIfAllowed;
        this.logIfDenied = logIfDenied;
        this.resourceReferenceCount = resourceReferenceCount;
    }

    /**
     * Resource on which action is being performed.
     */
    public ResourcePattern resourcePattern() {
        return resourcePattern;
    }

    /**
     * Operation being performed.
     */
    public AclOperation operation() {
        return operation;
    }

    /**
     * Indicates if audit logs tracking ALLOWED access should include this action if result is
     * ALLOWED. The flag is true if access to a resource is granted while processing the request as a
     * result of this authorization. The flag is false only for requests used to describe access where
     * no operation on the resource is actually performed based on the authorization result.
     */
    public boolean logIfAllowed() {
        return logIfAllowed;
    }

    /**
     * Indicates if audit logs tracking DENIED access should include this action if result is
     * DENIED. The flag is true if access to a resource was explicitly requested and request
     * is denied as a result of this authorization request. The flag is false if request was
     * filtering out authorized resources (e.g. to subscribe to regex pattern). The flag is also
     * false if this is an optional authorization where an alternative resource authorization is
     * applied if this fails (e.g. Cluster:Create which is subsequently overridden by Topic:Create).
     */
    public boolean logIfDenied() {
```

```
        return logIfDenied;
    }

    /**
     * Number of times the resource being authorized is referenced within the request. For example, a single
     * request may reference `n` topic partitions of the same topic. Brokers will authorize the topic once
     * with `resourceReferenceCount=n`. Authorizers may include the count in audit logs.
     */
    public int resourceReferenceCount() {
        return resourceReferenceCount;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Action)) {
            return false;
        }

        Action that = (Action) o;
        return Objects.equals(this.resourcePattern, that.resourcePattern) &&
            Objects.equals(this.operation, that.operation) &&
            this.resourceReferenceCount == that.resourceReferenceCount &&
            this.logIfAllowed == that.logIfAllowed &&
            this.logIfDenied == that.logIfDenied;

    }

    @Override
    public int hashCode() {
        return Objects.hash(resourcePattern, operation, resourceReferenceCount, logIfAllowed, logIfDenied);
    }

    @Override
    public String toString() {
        return "Action(" +
            ", resourcePattern='" + resourcePattern + '\'' +
            ", operation='" + operation + '\'' +
            ", resourceReferenceCount='" + resourceReferenceCount + '\'' +
            ", logIfAllowed='" + logIfAllowed + '\'' +
            ", logIfDenied='" + logIfDenied + '\'' +
            ')';
    }
}
```

Authorize method returns individual allowed/denied results for every action.

Authorizer Operation Results

```
package org.apache.kafka.server.authorizer;

public enum AuthorizationResult {
    ALLOWED,
    DENIED
}
```

ACL create operation returns any exception from each ACL binding requested.

Authorizer Operation Results

```
package org.apache.kafka.server.authorizer;

import java.util.Optional;
import org.apache.kafka.common.annotation.InterfaceStability;
import org.apache.kafka.common.errors.ApiException;

@InterfaceStability.Evolving
public class AclCreateResult {
    public static final AclCreateResult SUCCESS = new AclCreateResult();

    private final ApiException exception;

    private AclCreateResult() {
        this(null);
    }

    public AclCreateResult(ApiException exception) {
        this.exception = exception;
    }

    /**
     * Returns any exception during create. If exception is empty, the request has succeeded.
     */
    public Optional<ApiException> exception() {
        return exception == null ? Optional.empty() : Optional.of(exception);
    }
}
```

ACL delete operation returns any exception from each ACL filter requested. Matching ACL bindings for each filter are returned along with any delete failure.

Delete Results

```
package org.apache.kafka.server.authorizer;

import java.util.Collections;
import java.util.Collection;
import java.util.Optional;
import org.apache.kafka.common.acl.AclBinding;
import org.apache.kafka.common.annotation.InterfaceStability;
import org.apache.kafka.common.errors.ApiException;

@InterfaceStability.Evolving
public class AclDeleteResult {
    private final ApiException exception;
    private final Collection<AclBindingDeleteResult> aclBindingDeleteResults;

    public AclDeleteResult(ApiException exception) {
        this(Collections.emptySet(), exception);
    }

    public AclDeleteResult(Collection<AclBindingDeleteResult> deleteResults) {
        this(deleteResults, null);
    }

    private AclDeleteResult(Collection<AclBindingDeleteResult> deleteResults, ApiException exception) {
        this.aclBindingDeleteResults = deleteResults;
        this.exception = exception;
    }

    /**
     * Returns any exception while attempting to match ACL filter to delete ACLs.
     */
    public Optional<ApiException> exception() {
```

```
        return exception == null ? Optional.empty() : Optional.of(exception);
    }

    /**
     * Returns delete result for each matching ACL binding.
     */
    public Collection<AclBindingDeleteResult> aclBindingDeleteResults() {
        return aclBindingDeleteResults;
    }


    /**
     * Delete result for each ACL binding that matched a delete filter.
     */
    public static class AclBindingDeleteResult {
        private final AclBinding aclBinding;
        private final ApiException exception;

        public AclBindingDeleteResult(AclBinding aclBinding) {
            this(aclBinding, null);
        }

        public AclBindingDeleteResult(AclBinding aclBinding, ApiException exception) {
            this.aclBinding = aclBinding;
            this.exception = exception;
        }

        /**
         * Returns ACL binding that matched the delete filter. {@link #deleted()} indicates if
         * the binding was deleted.
         */
        public AclBinding aclBinding() {
            return aclBinding;
        }

        /**
         * Returns any exception that resulted in failure to delete ACL binding.
         */
        public Optional<ApiException> exception() {
            return exception == null ? Optional.empty() : Optional.of(exception);
        }
    }
}
```

Proposed Changes

Asynchronous update requests

`kafka.server.KafkaApis` will be updated to handle `CreateAcls` and `DeleteAcls` requests asynchronously using a purgatory. If `Authorizer.createAcls` or `Authorizer.deleteAcls` returns any `ComplettionStage` that is not complete, the request will be added to a purgatory and completed when all the stages complete. Authorizer implementations with low latency updates may continue to update synchronously and return a completed future. These requests will be completed in-line and will not be added to the purgatory.

Asynchronous updates are useful for Authorizer implementations that use external stores for ACLs, for example a database. Async handling of update requests will enable Kafka brokers to handle database outages without blocking request threads. As many databases now support async APIs (https://dev.mysql.com/doc/x-devapi-userguide/en/synchronous-vs-asynchronous-execution.html, https://blogs.oracle.com/java/jdbc-next:-a-new-asynchronous-api-for-connecting-to-a-database), async update API enables authorizers to take advantage of these APIs.

Purgatory metrics will be added for ACL updates, consistent with metrics from other purgatories. Two new metrics will be added:

- `kafka.server:type=DelayedOperationPurgatory,name=NumDelayedOperations,delayedOperation=acl-update`
- `kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=acl-update`

In addition to these metrics, existing request metrics for `CreateAcls` and `DeleteAcls` can be used to track the portion of time spent on async operations since local time is updated before the async wait and remote time is updated when async wait completes:

- `kafka.network:type=RequestMetrics,name=LocalTimeMs,request=CreateAcls`
- `kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=CreateAcls`

- kafka.network:type=RequestMetrics,name=LocalTimeMs,request=DeleteAcls
- kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=DeleteAcls

Deprecate existing Scala Authorizer Trait

kafka.security.auth.Authorizer will be deprecated along with all the supporting Scala classes including Resource, Operations and ResourceTypes. A new AuthorizerWrapper class will be introduced to wrap implementations using the Scala trait into the new Java interface. All usage of Authorizer (e.g. in KafkaApis) will be replaced with the new authorizer interface.

AclAuthorizer

A new authorizer implementation will be added. kafka.security.authorizer.AclAuthorizer will implement the new interface, making use of the additional request context available to improve authorization logging. This authorizer will be compatible with SimpleAclAuthorizer and will support all its existing configs including super.users.

SimpleAclAuthorizer

SimpleAclAuthorizer will be deprecated, but we will continue to support this implementation using the old API. Since it is part of the public API, all its public methods will be retained without change. This enables existing custom implementations that rely on this class to continue to be used.

Optional Interfaces

Reconfigurable

kafka.server.DynamicBrokerConfig will be updated to support dynamic update of authorizers which implement org.apache.kafka.common.Reconfigurable. Authorizer implementations can react to dynamic updates of any of its configs including custom configs, avoiding broker restarts to update configs.

Compatibility, Deprecation, and Migration Plan

What impact (if any) will there be on existing users?

Existing authorizer interfaces and classes are being deprecated, but not removed. We will continue to support the same config with the old API to ensure that existing users are not impacted.

If we are changing behavior how will we phase out the older behavior?

We are deprecating the existing Scala authorizer API. These will be removed in a future release, but will continue to be supported for backward compatibility until then. Until the old authorizer is removed, no config changes are required during upgrade.

Test Plan

All the existing integration and system tests will be updated to use the new config and the new authorization class. Unit tests will be added for testing the new methods and parameters being introduced in this KIP. An additional integration test will be added to test authorizers using the old Scala API.

Rejected Alternatives

Description of Authorizer as proposed in KIP-50

KIP-50 proposes to return a textual description of the authorizer that can be used in tools like AclCommand. Since we don't support returning a description using AdminClient and none of the other pluggable APIs have similar support, this KIP does not add a method to return authorizer description.

Separate authorizers for each listener

In some environments, authorization decisions may be dependent on the security protocol used by the client or the listener on which the request was received. We have listener prefixed configs to enable independent listener-specific configs for authentication etc. But since authorizers tend to cache a lot of metadata and need to watch for changes in metadata, a single shared instance works better for authorization. This KIP proposes a single authorizer that can use listener and security protocol provided in the authorization context to include listener-specific authorization logic.

Extend SimpleAclAuthorizer to support the new API

`SimpleAclAuthorizer` is part of our public API since it is in the public package `kafka.security.auth`. So we need to ensure that the old API is used with this authorizer if custom implementations extend this class and override specific methods. So this KIP deprecates, but retains this implementation and adds a separate implementation that uses the new API.

Make authorize() asynchronous

Authorize operations in the existing `Authorizer` are synchronous and this KIP proposes to continue to authorize synchronously on the request thread while processing each request. This requires all ACLs to be cached in every broker to avoid blocking request threads during authorization. To improve scalability in future, we may want to support asynchronous authorize operations that may perform remote communication, for example with an LRU cache. But asynchronous authorize operations add complexity to the Kafka implementation. Even though we may be able to use the existing purgatory, additional design work is required to figure out how this can be implemented efficiently. So it was decided that we should keep the authorization API synchronous for now. In future, we can add async authorize as a new method on the API if required.