KIP-285: Connect Rest Extension Plugin

Status

Current state: Accepted

Discussion thread: *here*

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

PR *: https://github.com/apache/kafka/pull/4931*

Released: 2.0.0

Motivation

Connect Framework offers REST API that is used to mange the lifecycle of the connector. Its imperative in most enterprises to secure the API and also add authorization to the end points. We could add the ability for authentication and authorization in the framework. But the security requirements are so broad that it's not practical to support all of them in the framework. Hence we must provide ability for users to plug resources that help achieve the required capabilities.

While security is prime use cases for this extension. Its not limited to that. Some of the common use cases are

- Build a custom Authentication filter
- Build a custom Authorization filter
- Complex extensions can even provide filters that rewrite/validate the connector requests to enforce additional constraints on the connector configurations

Public Interfaces

Developers would be required to implement only the **ConnectRestExtension** interface to provide an extension. ConnectRestExtension provides an implementation of ConnectRestExtensionContext whose configurable() provides an implementation of javax.ws.rs.core.Configurable. Using this developers can register new JAX-RS resources like filter, new end points, etc

```
package org.apache.kafka.connect.rest;
public interface ConnectRestExtension extends Configurable, Versioned, Closeable {
    /**
     * ConnectRestExtension implementations register custom JAX-RS resources via the {@link
     * #register(ConnectRestExtensionContext)} method. Framework will invoke this method after
     * registering the default Connect resources. If the implementations attempt to re-register any
     * of the Connect Resources, it will be be ignored and will be logged.
     *
     * @param restPluginContext The context provides access to JAX-RS {@link javax.ws.rs.core.Configurable}
     *                          and {@link ConnectClusterState}.The custom JAX-RS resources can be
     *                          registered via the {@link ConnectRestExtensionContext#configurable()}
     */
    void register(ConnectRestExtensionContext restPluginContext);
}
```

As mentioned above, even though the developers are required to only implement the **ConnectRestExtension,** they will be using several new public interfaces that are implemented by the framework.

Versioned

A new Versioned interface that will be used by all the plugins/components that support version. The Connector interface would be modified to extend this new interface instead of exposing the version() method itself.

```
package org.apache.kafka.connect.components;
public interface Versioned {
    /**
     * Get the version of this component.
     *
     * @return the version, formatted as a String
     */
    String version();
}
```

ConnectRestExtensionContext

This is a request Context interface that composes and provides access to

- Configurable - register JAX-RS resources
- clusterState - A new interface that helps provide some cluster state information

```
package org.apache.kafka.connect.rest;
interface ConnectRestExtensionContext{
    /**
     *
     * @return return a  implementation of {@link javax.ws.rs.core.Configurable} that be used ot
     * register JAX-RS resources
     */
    Configurable<? extends Configurable> configurable();
    /**
     * Provides meta data about connector's and its health
     * @return instance of {@link ConnectClusterState}
     */
    ConnectClusterState clusterState();
}
```

ConnectClusterState

This interface provides methods for the extension to get the connector states and list of running connectors.

```
package org.apache.kafka.connect.health;
interface ConnectClusterState{
    /**
     * Get a list of connectors currently running in this cluster. This is a full list of connectors in the
```

```
cluster gathered
     * from the current configuration.
     */
    Collection<String> connectors();

    /**
     * Lookup the current status of a connector.
     * @param connName name of the connector
     */
    ConnectorHealth connectorHealth(String connName);

}

package org.apache.kafka.connect.health;
public class ConnectorHealth {

    private final String name;
    private final ConnectorState connector;
    private final Map<Integer, TaskState> tasks;
    private final ConnectorType type;


    public ConnectorHealth(String name,
                           ConnectorState connector,
                           Map<Integer, TaskState> tasks,
                           ConnectorType type) {
        this.name = name;
        this.connector = connector;
        this.tasks = tasks;
        this.type = type;
    }


    public String name() {
        return name;
    }


    public ConnectorState connectorState() {
        return connector;
    }


    public Map<Integer, TaskState> tasksState() {
        return tasks;
    }


    public ConnectorType type() {
        return type;
    }


}

public abstract class AbstractState {

    private final String state;
    private final String trace;
    private final String workerId;

    public AbstractState(String state, String workerId, String trace) {
        this.state = state;
        this.workerId = workerId;
        this.trace = trace;
    }

    public String state() {
        return state;
    }
```

```
    public String workerId() {
        return workerId;
    }

    public String trace() {
        return trace;
    }
}


public class ConnectorState extends AbstractState {

    public ConnectorState(String state, String worker, String msg) {
        super(state, worker, msg);
    }
}

public class TaskState extends AbstractState implements Comparable<TaskState> {

    private final int taskId;

    public TaskState(int taskId, String state, String workerId, String msg) {
        super(state, workerId, msg);
        this.taskId = taskId;
    }

    public int taskId() {
        return taskId;
    }

    @Override
    public int compareTo(TaskState that) {
        return Integer.compare(this.taskId, that.taskId);
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (!(o instanceof TaskState)) {
            return false;
        }
        TaskState other = (TaskState) o;
        return compareTo(other) == 0;
    }

    @Override
    public int hashCode() {
        return Objects.hash(taskId);
    }
}
```

This also introduces a new configuration that rest.extension.classes that allows to configure a comma separated list of Rest extension implementations.

Proposed Changes

Plugin Interface

Users will be able to create a plugin by implementing the **ConnectRestExtension** interface, which has a single method that takes a **ConnectRestExtensionContext** instance as the only parameter. This allows us to change the interface easily in future to add new parameters. Connect runtime would also provide a default implementation for the interface **ConnectRestExtensionContext.** One or more of the **ConnectRestExtension** implementation can be configured via the configuration rest.extension.classes as a comma separated list of class names.

Implementations would use the `javax.ws.rs.core.Configurable` to register one or more JAX-RS resources and get access to the Worker's Configs through the configure(Map<String, ?> configs) method in the ConnectRestExtension implementation(through org.apache.kafka.common.Configurable)

```
package org.apache.kafka.connect.runtime.rest;
class ConnectRestExtensionContextImpl implements ConnectRestExtensionContext{
    private final Configurable configurable;
    private final ConnectClusterState clusterState;

    ConnectRestExtensionContext(Configurable configurable, ConnectClusterState clusterState){
        this.configurable = configurable;
        this.clusterState = clusterState;
    }

    public Configurable configurable(){
        return this.configurable;
    }

    public ConnectClusterState clusterState(){
        return this.clusterState;
    }
}
```

We will be introducing another new public API ConnectClusterState which will at present provide some of the read only methods from the Herder. The change would also include a default implementation ConnectClusterStateImpl in the connect runtime that will delegate to the underlying Herder. This will be useful when you want to add new resources like healthcheck, monitoring, etc.

```
package org.apache.kafka.connect.runtime.health;
class ConnectClusterStateImpl implements ConnectClusterState{
    private final Herder herder;

    public ConnectClusterStateImpl(Herder herder){
        this.herder = herder;
    }

    @Override
    Collection<String> connectors(){
        //delegate to herder
    }

    @Override
    ConnectorHealth connectorHealth(String connName);{
        //delegate to herder
    }
}
```

Rest Extension Integration with Connect

The plugin's would be registered in the RestServer.start(Herder herder) method after registering the default Connect resources. Connect Runtime would provide an implementation of Configurable interface that would do the following.

- Constructed with the ResourceConfig available in the RestServer and the configure(Map<String, ?> configs) is invoked on the implementation
- Will check if resource is already registered. If not, it would delegate to ResourceConfig. If already registered would log a warning message.
- For non-register methods would just delegate to the ResourceConfig instance. This helps alleviate any issues that could arise if Extension accidentally reregister the connect resources.
- The close() for the plugins would be invoked as part of the stop() in the RestServer. The implementation would not be invoked after this.

```
class ConnectRestConfigurable implements Configurable{
        ResourceConfig resourceConfig;

        public ConnectRestConfigurable(ResourceConfig resourceConfig) {
                this.resourceConfig = resourceConfig;
        }

        //implement methods and delegate to resourceConfig
}
```

Packaging

The new extension class and its dependencies would need to be as part of the plugin path. Hence ConnectRestExtension would be defined as a new plugin to be loaded by the PluginClassLoader.The plugin would be looked up based on Java's Service Provider API instead of the Reflections scan that is used for other plugins. This will help in terms of not adding class loader cost that is associated in scanning the classes today for other plugins. Hence the implementation must provide a `META-INF/services/org.apache.kafka.connect.rest.ConnectRestExtension` as part of the jar file containing the fully qualified implementation class .

Example

Consider the following example that defines a single plugin to add an authenticating filter and a health check resource.

```
class  ExampleConnectRestExtension implements ConnectRestExtension{

        private Map<String, ?> configs;

        @Override
    public void register(ConnectRestExtensionContext restPluginContext){
        restPluginContext.configurable().register(new AuthenticationFilter(configs));
        restPluginContext.configurable().register(new HealthCheckResource(configs, restPluginContext.
clusterState()));
    }

    @Override
    public void close() throws IOException {
    }

    @Override
    public void configure(Map<String, ?> configs) {
                this.configs = configs;
    }

    @Override
    public String version() {
        return AppInfoParser.getVersion();
    }
}

class AuthenticationFilter implements ContainerRequestFilter {
 private final String authenticationRealm;
  public AuthenticationFilter(Map<String, ?> configs){
    //set up filter
        authenticationRealm = configs.get("example.authentication.realm");
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
        //authentication logic
  }

}

@Path("/connect")
class HealthCheckResource {

    public HealthCheckResource(Map<String, ?> configs, ConnectClusterState clusterState){
        //initialize resource
    }

    @path("/health")
    public void healthcheck(){
        //check herder health
    }
}
```

For the RestExtension implementation to be found, the JAR should include the classes required by the implementation (excluding any Connect API or JAX-RS JARs) and should include a META-INF/services/org.apache.kafka.connect.rest.ConnectRestExtension file that contains the fully-qualified names of the extension implementation class(es) found within the JAR. This is the standard Java Service Loader API mechanism.

> ***META-INF/services/org.apache.kafka.connect.rest.ConnectRestException***
>
> *com.example.ExampleConnectRestExtension*

The above illustrated plugin can then be configured in the worker's configuration as below

> ***worker.properties***

```
rest.extension.classes=com.example.ExampleConnectPlugin
example.authentication.realm=ExampleAuthentication
```

Reference Implementation

The KIP proposes to include a reference implementation that allows users to authenticate incoming Basic Auth headers against the configured JAASLoginModule.

Compatibility, Deprecation, and Migration Plan

- This is entirely new functionality so there are no compatibility, deprecation, or migration concerns.

Rejected Alternatives

1. Creating configs specific to the plugin and just passing them to the plugin based on a prefix. It was considered much easier to make the complete WorkerConfig. Also, in many cases the plugins would need to know just more than their configs to implement their actions.
2. Passing the Herder to the plugin was considered but it was rejected since the Herder API is not public and we don't want to expose the complete Herder capabilities to the plugin.
3. Providing ability to just add Filters insteda of any kind of Jersey resource was considered but it was rejected because it was too limiting in its capability that one canot add new resource end points or add a jersey provider.
4. Having the Connect REST plugin as part of class path is rejected for the same reason why we have custom interfaces like Converters and Connectors as plugin and loaded via plugin path.