package com.twitter.search.earlybird\_root.filters;

import java.util.List;

import javax.inject.Inject;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.finagle.Service;

import com.twitter.finagle.SimpleFilter;

import com.twitter.search.common.decider.SearchDecider;

import com.twitter.search.common.metrics.SearchCounter;

import com.twitter.search.earlybird.thrift.EarlybirdDebugInfo;

import com.twitter.search.earlybird.thrift.EarlybirdRequest;

import com.twitter.search.earlybird.thrift.EarlybirdResponse;

import com.twitter.search.earlybird.thrift.EarlybirdResponseCode;

import com.twitter.search.earlybird\_root.common.EarlybirdRequestContext;

import com.twitter.search.queryparser.query.Query;

import com.twitter.search.queryparser.query.QueryNodeUtils;

import com.twitter.search.queryparser.query.QueryParserException;

import com.twitter.search.queryparser.query.search.SearchOperator;

import com.twitter.search.queryparser.query.search.SearchOperatorConstants;

import com.twitter.search.queryparser.visitors.DropAllProtectedOperatorVisitor;

import com.twitter.search.queryparser.visitors.QueryTreeIndex;

import com.twitter.util.Future;

/\*\*

\* Full archive service filter validates requests with a protected operator, appends the

\* '[exclude protected]' operator by default, and appends '[filter protected]' operator instead if

\* 'getProtectedTweetsOnly' request param is set. A client error response is returned if any of the

\* following rules is violated.

\* 1. There is at most one 'protected' operator in the query.

\* 2. If there is a 'protected' operator, it must be in the query root node.

\* 3. The parent node of the 'protected' operator must not be negated and must be a conjunction.

\* 4. If there is a positive 'protected' operator, 'followedUserIds' and 'searcherId' request

\* params must be set.

\*/

public class FullArchiveProtectedOperatorFilter extends

SimpleFilter<EarlybirdRequestContext, EarlybirdResponse> {

private static final Logger LOG =

LoggerFactory.getLogger(FullArchiveProtectedOperatorFilter.class);

private static final SearchOperator EXCLUDE\_PROTECTED\_OPERATOR =

new SearchOperator(SearchOperator.Type.EXCLUDE, SearchOperatorConstants.PROTECTED);

private static final SearchOperator FILTER\_PROTECTED\_OPERATOR =

new SearchOperator(SearchOperator.Type.FILTER, SearchOperatorConstants.PROTECTED);

private static final SearchCounter QUERY\_PARSER\_FAILURE\_COUNT =

SearchCounter.export("protected\_operator\_filter\_query\_parser\_failure\_count");

private final DropAllProtectedOperatorVisitor dropProtectedOperatorVisitor;

private final SearchDecider decider;

@Inject

public FullArchiveProtectedOperatorFilter(

DropAllProtectedOperatorVisitor dropProtectedOperatorVisitor,

SearchDecider decider) {

this.dropProtectedOperatorVisitor = dropProtectedOperatorVisitor;

this.decider = decider;

}

@Override

public Future<EarlybirdResponse> apply(

EarlybirdRequestContext requestContext,

Service<EarlybirdRequestContext, EarlybirdResponse> service) {

Query query = requestContext.getParsedQuery();

if (query == null) {

return service.apply(requestContext);

}

QueryTreeIndex queryTreeIndex = QueryTreeIndex.buildFor(query);

List<Query> nodeList = queryTreeIndex.getNodeList();

// try to find a protected operator, returns error response if more than one protected

// operator is detected

SearchOperator protectedOperator = null;

for (Query node : nodeList) {

if (node instanceof SearchOperator) {

SearchOperator searchOp = (SearchOperator) node;

if (SearchOperatorConstants.PROTECTED.equals(searchOp.getOperand())) {

if (protectedOperator == null) {

protectedOperator = searchOp;

} else {

return createErrorResponse("Only one 'protected' operator is expected.");

}

}

}

}

Query processedQuery;

if (protectedOperator == null) {

// no protected operator is detected, append '[exclude protected]' by default

processedQuery = QueryNodeUtils.appendAsConjunction(query, EXCLUDE\_PROTECTED\_OPERATOR);

} else {

// protected operator must be in the query root node

if (queryTreeIndex.getParentOf(protectedOperator) != query) {

return createErrorResponse("'protected' operator must be in the query root node");

}

// the query node that contains protected operator must not be negated

if (query.mustNotOccur()) {

return createErrorResponse("The query node that contains a 'protected' operator must not"

+ " be negated.");

}

// the query node that contains protected operator must be a conjunction

if (!query.isTypeOf(Query.QueryType.CONJUNCTION)) {

return createErrorResponse("The query node that contains a 'protected' operator must"

+ " be a conjunction.");

}

// check the existence of 'followedUserIds' and 'searcherId' if it is a positive operator

if (isPositive(protectedOperator)) {

if (!validateRequestParam(requestContext.getRequest())) {

return createErrorResponse("'followedUserIds' and 'searcherId' are required "

+ "by positive 'protected' operator.");

}

}

processedQuery = query;

}

// update processedQuery if 'getProtectedTweetsOnly' is set to true, it takes precedence over

// the existing protected operators

if (requestContext.getRequest().isGetProtectedTweetsOnly()) {

if (!validateRequestParam(requestContext.getRequest())) {

return createErrorResponse("'followedUserIds' and 'searcherId' are required "

+ "when 'getProtectedTweetsOnly' is set to true.");

}

try {

processedQuery = processedQuery.accept(dropProtectedOperatorVisitor);

} catch (QueryParserException e) {

// this should not happen since we already have a parsed query

QUERY\_PARSER\_FAILURE\_COUNT.increment();

LOG.warn(

"Failed to drop protected operator for serialized query: " + query.serialize(), e);

}

processedQuery =

QueryNodeUtils.appendAsConjunction(processedQuery, FILTER\_PROTECTED\_OPERATOR);

}

if (processedQuery == query) {

return service.apply(requestContext);

} else {

EarlybirdRequestContext clonedRequestContext =

EarlybirdRequestContext.copyRequestContext(requestContext, processedQuery);

return service.apply(clonedRequestContext);

}

}

private boolean validateRequestParam(EarlybirdRequest request) {

List<Long> followedUserIds = request.followedUserIds;

Long searcherId = (request.searchQuery != null && request.searchQuery.isSetSearcherId())

? request.searchQuery.getSearcherId() : null;

return followedUserIds != null && !followedUserIds.isEmpty() && searcherId != null;

}

private boolean isPositive(SearchOperator searchOp) {

boolean isNegateExclude = searchOp.mustNotOccur()

&& searchOp.getOperatorType() == SearchOperator.Type.EXCLUDE;

boolean isPositive = !searchOp.mustNotOccur()

&& (searchOp.getOperatorType() == SearchOperator.Type.INCLUDE

|| searchOp.getOperatorType() == SearchOperator.Type.FILTER);

return isNegateExclude || isPositive;

}

private Future<EarlybirdResponse> createErrorResponse(String errorMsg) {

EarlybirdResponse response = new EarlybirdResponse(EarlybirdResponseCode.CLIENT\_ERROR, 0);

response.setDebugInfo(new EarlybirdDebugInfo().setHost("full\_archive\_root"));

response.setDebugString(errorMsg);

return Future.value(response);

}

}