Background

According to the description for solution 1 in the document:
https://docs.google.com/document/d/1_2U_k33memegm-jEjtgDULUUdkaSoe3UpJBW1oiAJgE, we consider adding a **getFilesetContext** interface on the server side. Using this interface:
1. We can report some data operation information in GVFS to the server side to audit.
2. We can maintain the logic of converting and obtaining the actual storage path, which originally needed to be maintained in multiple clients(Hadoop / Python GVFS), on the server side, reducing the complexity and maintenance difficulty of the client logic.
This document mainly considers and discusses how to pass these audit information to the server in the most appropriate way.

Research

1. Using HTTP Request Body or HTTP Header to pass the audit information?

Using HTTP Request Body

Solution 1: Define some audit information in the API module

By adding some enumeration classes and interfaces in the API module to specify the audit information that needs to be reported.

1. API Module:

```
public interface FilesetDataOperationCtx {
 String subPath();

 FilesetDataOperation operation();

 ClientType clientType();
}
```

```
public enum FilesetDataOperation {
 CREATE,
 OPEN,
 APPEND,
 RENAME,
 DELETE,
 GET_FILE_STATUS,
 LIST_STATUS,
 MKDIRS,
 GET_DEFAULT_REPLICATION,
 GET_DEFAULT_BLOCK_SIZE,
 SET_WORKING_DIR,
 UNKNOWN;
}
```

```
public enum ClientType {
 HADOOP_GVFS,
 PYTHON_GVFS,
 UNKNOWN;
}
```

Then we add the calling logic of the **getFilesetContext** interface on the server and client respectively, so as to call the interface in GVFS to complete the information reporting.

2. Server side:

```
@POST
@Path("{fileset}/context")
public Response getFilesetContext(
   @PathParam("metalake") String metalake,
   @PathParam("catalog") String catalog,
   @PathParam("schema") String schema,
   @PathParam("fileset") String fileset,
   GetFilesetContextRequest request) {
 LOG.info(
     "Received get fileset context request: {}.{}.{}.{}", metalake,
catalog, schema, fileset);
 try {
   return Utils.doAs(
       httpRequest,
       () -> {
         NameIdentifier ident = NameIdentifierUtil.ofFileset(metalake,
catalog, schema, fileset);
         BaseFilesetDataOperationCtx ctx =
             BaseFilesetDataOperationCtx.builder()
                 .withSubPath(request.getSubPath())
                 .withOperation(request.getOperation())
                 .withClientType(request.getClientType())
                 .build();
         FilesetContext context =
             TreeLockUtils.doWithTreeLock(
                 ident, LockType.READ, () ->
dispatcher.getFilesetContext(ident, ctx));
         return Utils.ok(new
FilesetContextResponse(DTOConverters.toDTO(context)));
       });
 } catch (Exception e) {
   return ExceptionHandlers.handleFilesetException(OperationType.GET,
fileset, schema, e);
 }
}
```

3. Java Client side:

```
@Override
public FilesetContext getFilesetContext(NameIdentifier ident,
FilesetDataOperationCtx ctx)
   throws NoSuchFilesetException {
 checkFilesetNameIdentifier(ident);
 Namespace fullNamespace = getFilesetFullNamespace(ident.namespace());
 GetFilesetContextRequest req =
     GetFilesetContextRequest.builder()
         .subPath(ctx.subPath())
         .operation(ctx.operation())
         .clientType(ctx.clientType())
         .build();
 FilesetContextResponse resp =
     restClient.post(
         formatFilesetRequestPath(fullNamespace) + "/" + ident.name() +
"/" + "context",
         req,
         FilesetContextResponse.class,
         Collections.emptyMap(),
         ErrorHandlers.filesetErrorHandler());
 resp.validate();

 return resp.getContext();
}
```

4. Common module:

```
public class GetFilesetContextRequest implements RESTRequest {
 @JsonProperty("subPath")
 private String subPath;

 @JsonProperty("operation")
 private FilesetDataOperation operation;

 @JsonProperty("clientType")
 private ClientType clientType;

 @Override
 public void validate() throws IllegalArgumentException {
   Preconditions.checkArgument(subPath != null, "\"subPath\" field cannot
be null");
   Preconditions.checkArgument(
       operation != null, "\"operation\" field is required and cannot be
null");
   Preconditions.checkArgument(
       clientType != null, "\"clientType\" field is required and cannot be
null");
 }
}
```

Pros

1. **Standardized**: For service calls(like Hadoop / Python GVFS), the parameters are more standardized.
2. **Pre-check**: Some parameter checks can be performed in advance when building parameters.

Cons

1. **Lack of scalability**: The inevitable result of standardization is that scalability becomes worse. Every time a new parameter is added, the API module needs to be modified and republished.
2. **Confusing for regular users**: Since these enumeration classes and interfaces are defined in the API module and will be exposed in the Java/Python Client later, it may be difficult for users to understand their meaning because they are only valid for clients or applications such as Hadoop / Python GVFS.

Solution 2: Use **Map<String, String>** to pass the audit information

This solution will no longer expose context content in the API module. When the client interacts with the server, a **Map<String, String>** serialized by Json is passed in the Request Body to save the context information.

1. Server side

```
@POST
@Path("{fileset}/context")
@ResponseMetered(name = "get-fileset-context", absolute = true)
public Response getFilesetContext(
   @PathParam("metalake") String metalake,
   @PathParam("catalog") String catalog,
   @PathParam("schema") String schema,
   @PathParam("fileset") String fileset,
   GetFilesetContextRequest request) {
 LOG.info(
     "Received get fileset context request: {}.{}.{}.{}", metalake,
catalog, schema, fileset);
 try {
   return Utils.doAs(
       httpRequest,
       () -> {
         NameIdentifier ident = NameIdentifierUtil.ofFileset(metalake,
catalog, schema, fileset);
         FilesetContext context =
             TreeLockUtils.doWithTreeLock(
                 ident, LockType.READ, () ->
dispatcher.getFilesetContext(ident, request.getSubPath(),
request.getContext()));
         return Utils.ok(new
FilesetContextResponse(DTOConverters.toDTO(context)));
       });
 } catch (Exception e) {
   return ExceptionHandlers.handleFilesetException(OperationType.GET,
fileset, schema, e);
 }
}
```

2. Java Client side:

```
@Override
public FilesetContext getFilesetContext(NameIdentifier ident, String
subPath, Map<String, String> context)
   throws NoSuchFilesetException {
 checkFilesetNameIdentifier(ident);
 Namespace fullNamespace = getFilesetFullNamespace(ident.namespace());
 GetFilesetContextRequest req =
     GetFilesetContextRequest.builder()
         .subPath(subPath)
         .context(context)
         .build();

 FilesetContextResponse resp =
     restClient.post(
         formatFilesetRequestPath(fullNamespace) + "/" + ident.name() +
"/" + "context",
         req,
         FilesetContextResponse.class,
         Collections.emptyMap(),
         ErrorHandlers.filesetErrorHandler());
 resp.validate();

 return resp.getContext();
}
```

3. Common module:

```
public class GetFilesetContextRequest implements RESTRequest {
 @JsonProperty("subPath")
 private String subPath;

 @JsonProperty("context")
 private Map<String, String> context;

 @Override
 public void validate() throws IllegalArgumentException {
   Preconditions.checkArgument(subPath != null, "\"subPath\" field cannot
be null");
 }
}
```

Pros

1. **Hide context details**: This solution still exposes the API in the Java Client, but compared to solution 1, it uses **Map<String, String>** to store context, which can hide the details of the context from the user.
2. **Strong scalability**: Since **Map<String, String>** is used to store context, it has strong scalability for service auditing. For example, when calling in GVFS, we can only add some information that needs to be reported in GVFS, and on the server side, the entire Map can be serialized and saved in the audit log.
3. **Can support larger context content**: In HTTP requests, using Request Body to pass data can support a larger size, while Header often limits the size to a smaller size.

Cons

1. **Unable to restrict user input**: Using Map means that user input cannot be restricted, which may cause users to enter irrelevant content when the API is exposed to users, ultimately affecting the service audit content.
2. **Lack of reusability**: Although only this one API needs to pass the context, it is difficult to avoid that other APIs will also need to pass the context in the future. Maintaining it in the Request Body will make it difficult to reuse it in other APIs.

Using HTTP Headers

Currently, it is a common solution for passing the audit information in the HTTP headers in some mature projects, such as:

- HDFS client and NN using CallerContext to pass the audit info in the request header: https://issues.apache.org/jira/browse/HDFS-9184

- Skywalking which is the popular distributed tracing system supports using header to pass the trace info and extend info: https://skywalking.apache.org/docs/main/latest/en/api/x-process-propagation-headers-v3/
- Zipkin another popular distributed tracing system uses the header to pass the trace info: https://zipkin.io/pages/instrumenting.html
- Google Healthcare API supports that user can custom the HTTP headers for audit logs: https://cloud.google.com/healthcare-api/docs/audit-log-http-headers
- Azure FHIR service supports that user can custom the HTTP headers for audit logs: https://learn.microsoft.com/en-us/azure/healthcare-apis/fhir/use-custom-headers-diagnosticlog

Therefore, it seems to be a better choice to put the audit information in the HTTP Header due to a large number of open source and commercial product practices.

1. API Module

```
FilesetContext getFilesetContext(NameIdentifier ident, String subPath)
   throws NoSuchFilesetException;
```

2. Server side

```
@GET
@Path("{fileset}/context")
@ResponseMetered(name = "get-fileset-context", absolute = true)
public Response getFilesetContext(
   @PathParam("metalake") String metalake,
   @PathParam("catalog") String catalog,
   @PathParam("schema") String schema,
   @PathParam("fileset") String fileset,
   @QueryParam("subPath") String subPath) {
 LOG.info(
     "Received get fileset context request: {}.{}.{}.{}", metalake,
catalog, schema, fileset);
 try {
   return Utils.doAs(
       httpRequest,
       () -> {
         NameIdentifier ident = NameIdentifierUtil.ofFileset(metalake,
catalog, schema, fileset);
         Map<String, String> filteredAuditMap =
Utils.filterAuditHeaders(httpRequest);
         InternalCallerContext internalCallerContext =
InternalCallerContext.builder()
                 .withContext(filteredAuditMap)
                 .build();

InternalCallerContext.InternalCallerContextHolder.set(internalCallerContext
);
         FilesetContext filesetContext =
             TreeLockUtils.doWithTreeLock(
                 ident, LockType.READ, () ->
dispatcher.getFilesetContext(ident, subPath));
         return Utils.ok(new
FilesetContextResponse(DTOConverters.toDTO(filesetContext)));
       });
 } catch (Exception e) {
   return ExceptionHandlers.handleFilesetException(OperationType.GET,
fileset, schema, e);
 }
}
```

```java
public class FilesetEventDispatcher implements FilesetDispatcher {

public FilesetContext getFilesetContext(NameIdentifier ident, String
subPath)
   throws NoSuchFilesetException {
 try {
   FilesetContext context = dispatcher.getFilesetContext(ident, subPath);
   InternalCallerContext internalContext =
InternalCallerContext.InternalCallerContextHolder.get();
   eventBus.dispatchEvent(
       new GetFilesetContextEvent(PrincipalUtils.getCurrentUserName(),
ident, subPath, internalContext));
   return context;
 } catch (Exception e) {
   eventBus.dispatchEvent(
       new GetFilesetContextFailureEvent(
           PrincipalUtils.getCurrentUserName(), ident, subPath, e));
   throw e;
 }
}
}
```

```java
public final class GetFilesetContextEvent extends FilesetEvent {
 private final String subPath;
 private final InternalCallerContext context;

 /**
  * Constructs a new {@code GetFilesetContextEvent}, recording the
attempt to get a fileset
  * context.
  *
  * @param user The user who initiated the get fileset context operation.
  * @param identifier The identifier of the fileset context that was
attempted to be got.
  * @param ctx The data operation context to get the fileset context.
  */
 public GetFilesetContextEvent(String user, NameIdentifier identifier,
String subPath, InternalCallerContext context) {
   super(user, identifier);
   this.subPath = subPath;
   this.context = context;
 }

 public String subPath() {
   return subPath;
 }

 public InternalCallerContext context() {
   return context;
 }
}
```

3. Client side

```java
@Override
public FilesetContext getFilesetContext(NameIdentifier ident, String
subPath)
   throws NoSuchFilesetException {
checkFilesetNameIdentifier(ident);
Namespace fullNamespace = getFilesetFullNamespace(ident.namespace());
InternalCallerContext context =
InternalCallerContext.InternalCallerContextHolder.get();
FilesetContextResponse resp =
   restClient.get(
       formatFilesetRequestPath(fullNamespace)
               + "/" + ident.name()
               + "/" + "context?"
               + "subPath=" + subPath,
       FilesetContextResponse.class,
       context.contextMap(),
       ErrorHandlers.filesetErrorHandler());
resp.validate();

return resp.getContext();
}
```

4. Common module

```
public class InternalCallerContext {
 private Map<String, String> contextMap;

 public Map<String, String> contextMap() {
   return contextMap;
 }

 private static class Builder {
   private InternalCallerContext internalContext;

   private Builder() {
     internalContext = new InternalCallerContext();
   }

   public InternalCallerContext.Builder withContext(Map<String, String>
contextMap) {
     internalContext.contextMap = contextMap;
     return this;
   }

   private void validate() {
     Preconditions.checkArgument(internalContext.contextMap != null,
"contextMap cannot be null");
   }

   public InternalCallerContext build() {
     validate();
     return internalContext;
   }
 }

 public static InternalCallerContext.Builder builder() {
   return new InternalCallerContext.Builder();
 }

 public static class InternalCallerContextHolder {

   private static final ThreadLocal<InternalCallerContext> CONTEXT = new
ThreadLocal<>();
```

```
   public static InternalCallerContext get() {
     return CONTEXT.get();
   }

   public static void set(InternalCallerContext context) {
     CONTEXT.set(context);
   }

   public static void remove() {
     CONTEXT.remove();
   }
 }
}
```

Pros

1. **More widespread use**: There are a lot of open source and commercial product practices.
2. **Strong scalability**: Individual APIs can pass custom HTTP headers..
3. **User-friendly**: Ordinary users do not need to perceive.

Cons

1. **Size limit**: HTTP Headers should not set too many parameters.

2. Should we expose the getFilesetContext API facing other clients(Like Hadoop / Python GVFS) in a user-facing client, such as the Java / Python Client?

I have no particular preference for this, but if we want to reduce the confusion for users, my suggestion is that we can consider not exposing the API in user-facing clients, and let the clients that need to call this API maintain the corresponding logic themselves.
If we adopt this solution, we only need to maintain the corresponding logic in the server and Hadoop / Python GVFS without modifying the Java / Python Client logic.

1. API Module

```
default FilesetContext getFilesetContext(NameIdentifier ident, String
subPath)
   throws NoSuchFilesetException {
 throw new UnsupportedOperationException("Not implemented");
}
```

2. Server side

```
@GET
@Path("{fileset}/context")
@ResponseMetered(name = "get-fileset-context", absolute = true)
public Response getFilesetContext(
   @PathParam("metalake") String metalake,
   @PathParam("catalog") String catalog,
   @PathParam("schema") String schema,
   @PathParam("fileset") String fileset,
   @QueryParam("subPath") String subPath) {
 LOG.info(
     "Received get fileset context request: {}.{}.{}.{}", metalake,
catalog, schema, fileset);
 try {
   return Utils.doAs(
       httpRequest,
       () -> {
         NameIdentifier ident = NameIdentifierUtil.ofFileset(metalake,
catalog, schema, fileset);
         Map<String, String> filteredAuditMap =
Utils.filterAuditHeaders(httpRequest);
         InternalCallerContext internalCallerContext =
InternalCallerContext.builder()
                 .withContext(filteredAuditMap)
                 .build();

InternalCallerContext.InternalCallerContextHolder.set(internalCallerContext
);
         FilesetContext filesetContext =
             TreeLockUtils.doWithTreeLock(
                 ident, LockType.READ, () ->
dispatcher.getFilesetContext(ident, subPath));
         return Utils.ok(new
FilesetContextResponse(DTOConverters.toDTO(filesetContext)));
       });
 } catch (Exception e) {
   return ExceptionHandlers.handleFilesetException(OperationType.GET,
fileset, schema, e);
 }
```

```
public class FilesetEventDispatcher implements FilesetDispatcher {

public FilesetContext getFilesetContext(NameIdentifier ident, String
subPath)
   throws NoSuchFilesetException {
 try {
   FilesetContext context = dispatcher.getFilesetContext(ident, subPath);
   InternalCallerContext internalContext =
InternalCallerContext.InternalCallerContextHolder.get();
   eventBus.dispatchEvent(
       new GetFilesetContextEvent(PrincipalUtils.getCurrentUserName(),
ident, subPath, internalContext));
   return context;
 } catch (Exception e) {
   eventBus.dispatchEvent(
       new GetFilesetContextFailureEvent(
           PrincipalUtils.getCurrentUserName(), ident, subPath, e));
   throw e;
 }
}
}
```

```
public final class GetFilesetContextEvent extends FilesetEvent {
 private final String subPath;
 private final InternalCallerContext context;
 /**
  * Constructs a new {@code GetFilesetContextEvent}, recording the
attempt to get a fileset
  * context.
  *
  * @param user The user who initiated the get fileset context operation.
  * @param identifier The identifier of the fileset context that was
attempted to be got.
  * @param ctx The data operation context to get the fileset context.
  */
 public GetFilesetContextEvent(String user, NameIdentifier identifier,
String subPath, InternalCallerContext context) {
   super(user, identifier);
   this.subPath = subPath;
   this.context = context;
 }

 public String subPath() {
   return subPath;
 }

 public InternalCallerContext context() {
   return context;
 }
}
```

3. GVFS side

```
private FilesetContext getFilesetContext(NameIdentifier ident, String
subPath, Map<String, String> headers)
   throws NoSuchFilesetException {
checkFilesetNameIdentifier(ident);
Namespace fullNamespace = getFilesetFullNamespace(ident.namespace());
FilesetContextResponse resp =
   restClient.get(
       formatFilesetRequestPath(fullNamespace)
               + "/" + ident.name()
               + "/" + "context?"
               + "subPath=" + subPath,
       FilesetContextResponse.class,
       headers,
       ErrorHandlers.filesetErrorHandler());
resp.validate();
}
```

4. Common module

```
public class InternalCallerContext {
 private Map<String, String> contextMap;

 public Map<String, String> contextMap() {
   return contextMap;
 }

 private static class Builder {
   private InternalCallerContext internalContext;

   private Builder() {
     internalContext = new InternalCallerContext();
   }
   public InternalCallerContext.Builder withContext(Map<String, String>
contextMap) {
     internalContext.contextMap = contextMap;
     return this;
   }

   private void validate() {
     Preconditions.checkArgument(internalContext.contextMap != null,
"contextMap cannot be null");
   }

   public InternalCallerContext build() {
     validate();
     return internalContext;
   }
 }

 public static InternalCallerContext.Builder builder() {
   return new InternalCallerContext.Builder();
 }

 public static class InternalCallerContextHolder {

   private static final ThreadLocal<InternalCallerContext> CONTEXT = new
ThreadLocal<>();

   public static InternalCallerContext get() {
     return CONTEXT.get();
   }

   public static void set(InternalCallerContext context) {
     CONTEXT.set(context);
   }

   public static void remove() {
     CONTEXT.remove();
   }
 }
}
```