DOIP SDK v2 - Java™ Version

(Target audience for this document is Java Developers)

The DOIP Software Development Kit version 2 is intended to be used by Java developers to associate a DOIP version 2 interface to their system of choice. DOIP interface might be associated with Data Repositories and Metadata Registries, among other information systems.

The DOIP SDK Software consists of the DOIP server software as well as the DOIP client software. The DOIP server software converts DOIP network requests into Java requests and Java responses into DOIP network responses. The DOIP client software provides both TransportDoipClient, which includes Java classes that closely reflect the structures of the DOIP v2.0 specification, and DoipClient, which uses TransportDoipClient to expose a Java API that maps more directly with the conceptual usage of the DOIP with one Java method per basic operation.

Although a system implementing the DOIP interface is considered to play the role of a server, it might also play the role of a client if it intends to interact with other DOIP servers.

Please refer to the associated README for details on the SDK package.

DOIP Server Software

This section discusses how Java developers can implement a DOIP interface using the DOIP server software. The primary Java interface of relevance is DoipProcessor.

DoipProcessor

DoipProcessor is a Java interface that developers should implement to process incoming DOIP messages. In order to respond to DOIP requests, the 'process' method should be overridden. Below is an example Java class that implements DoipProcessor and simply echoes the bytes read from an incoming DoipServerRequest to a DoipServerResponse.

```
public class DoipEchoServer implements DoipProcessor {

  @Override
  public void process(DoipServerRequest req, DoipServerResponse resp) throws IOException {
    try {
      resp.commit();
      for (InDoipSegment segment : req.getInput()) {
        if (segment.isJson()) {
          resp.getOutput().writeJson(segment.getJson());
        } else {
          try (
              InputStream in = segment.getInputStream();
              OutputStream out = resp.getOutput().getBytesOutputStream();
          ) {
            byte[] buf = new byte[8192];
            int r;
            while ((r = in.read(buf)) > 0) {
              out.write(buf, 0, r);
            }
          }
        }
      }
```

```
        } catch (UncheckedIOException e) {
            throw e.getCause();
        }
    }
}
```

The DoipProcesser will be called from a multi-threaded setup. As such care must be taken to ensure synchronized access to shared resources.

It is the responsibility of the developer to read the input from the DoipServerRequest and write their response to the DoipServerResponse. Those request and response classes provide an abstraction over the DOIP segments such that the developer does not need to read and parse the raw InputStream of the DOIP protocol.

Specific headers can be extracted from the DoipServerRequest with getClientId(), getTargetId() and getOperationId(). Attributes can be retrieved as a JsonObject with getAttributes().

Below is an example that demonstrates branching based on operationId retrieved from the DoipServerRequest. Constants for common operation identifiers are located in DoipConstants.

```
@Override
public void process(DoipServerRequest req, DoipServerResponse resp) throws IOException {
  String operationId = req.getOperationId();
  String targetId = req.getTargetId();
  if (DoipConstants.OP_RETRIEVE.equals(operationId)) {
     retrieve(req, resp);
  } else if (DoipConstants.OP_UPDATE.equals(operationId)) {
     update(req, resp);
  } else if (DoipConstants.OP_DELETE.equals(operationId)) {
     delete(req, resp);
  } else if (DoipConstants.OP_LIST_OPERATIONS.equals(operationId)) {
     listOperations(targetId, req, resp);
  } else {
     resp.setStatus(DoipConstants.STATUS_DECLINED);
     resp.setAttribute(DoipConstants.MESSAGE_ATT, "Operation not supported");
  }
}
```

The segments of the request can be accessed with getInput(). getInput() returns an InDoipMessage. The specific implementation of this class can be used to retrieve the JSON and binary segments. For example, the JSON and binary segments of an operation input of type ‘update’ or ‘create’ could be read from the request as follows:

```
JsonObject json;
Map<String, InputStream> inputStreams = new HashMap<>();

InDoipMessage input = req.getInput();
InDoipSegment firstSegment = InDoipMessageUtil.getInitialSegment(input);
json = firstSegment.getJson().getAsJsonObject();

Iterator<InDoipSegment> segments = input.iterator();
while (segments.hasNext()) {
   InDoipSegment headerSegment = segments.next();
   String elementId;
   elementId = headerSegment.getJson().getAsJsonObject().get("id").getAsString();
   InDoipSegment elementBytesSegment = segments.next();
   inputStreams.put(elementId, elementBytesSegment.getInputStream());
```

```
}
```

Authentication

DoipProcessor does not implement any authentication logic. That is left as an exercise for the developer. However the DoipServerRequest has a method getAuthentication() for extracting the parsed authentication information from the request.

DoipServer

DoipServer is used to construct an instance of your custom DoipProcessor, listen for client connections on a TCP/IP network over a TLS channel, and invoke the process method on the DoipProcessor to process the request. It is constructed with a DoipServerConfig that, at a minimum, must specify the IP address and port to listen on and the name of the custom DoipProcessor class that should respond to requests.

```
public static void main(String[] args) throws Exception {
    DoipServerConfig config = new DoipServerConfig();
    config.listenAddress = "10.0.1.1";
    config.port = 8888;
    config.processorClass = DoipEchoServer.class.getName();
    DoipServer server = new DoipServer(config);
    server.init();
    Runtime.getRuntime().addShutdownHook(new Thread(server::shutdown));
}
```

DOIP Client Software

This section only discusses the DoipClient. Please refer to Javadoc for details on TransportDoipClient.

DoipClient

The client serves two purposes. First, it provides a Java API to invoke the basic operations as well as any extended operations, abstracting away the protocol-specific serialization details. And secondly, it uses the Handle System to discover where to send DOIP requests at request time using the target identifier of the digital object being interacted with. Additionally, the client maintains connection pools for the DOIP services being interacted with.

Constructing an instance of the DoipClient is done with a no-args constructor. There is no need to specify the IP address or port of the DOIP server to be communicate with as that information is discovered automatically when the request is made. However, should there be a need to explicitly instruct the client to communicate with a specific DOIP service, ServiceInfo can be optionally supplied with each request. An example to create a digital object is shown below:

```
DoipClient client = new DoipClient();
AuthenticationInfo authInfo = new PasswordAuthenticationInfo("admin", "password");

DigitalObject dobj = new DigitalObject();
dobj.id = "35.TEST/ABC";
dobj.type = "Document";
JsonObject content = new JsonObject();
content.addProperty("name", "example");
```

```
dobj.setAttribute("content", content);

Element el = new Element();
el.id = "file";
el.in = Files.newInputStream(Paths.get("/test.pdf"));
dobj.elements = new ArrayList<>();
dobj.elements.add(el);

ServiceInfo serviceInfo = new ServiceInfo("35.TEST/DOIPServer");

DigitalObject result = client.create(dobj, authInfo, serviceInfo);
```

ServiceInfo

The ServiceInfo contains information specifying where to send the request. This can be a handle that resolves to a handle record that contains a value of type DOIPServiceInfo, the structure of which is defined in the DOIP specification, or the ServiceInfo could directly contain the IP address, port and service identifier of the target service. In the above example a handle is supplied to direct the client to the service that the DigitalObject should be created at. A call to retrieve a DigitalObject with identifier that resolves to the service information would look like this:

```
DigitalObject result = client.retrieve("35.TEST/ABC", authInfo);
```

Here, only the handle of the object is passed into the call and the ServiceInfo is discovered by the client. Other operations that apply to existing objects could be invoked without supplying ServiceInfo. However, ServiceInfo can be supplied specifically as shown in the below example:

```
ServiceInfo serviceInfo = new ServiceInfo("35.TEST/DOIPServer", "10.0.1.1", 8888);
DigitalObject result = client.retrieve("35.TEST/ABC", authInfo, serviceInfo);
```

AuthenticationInfo

Three classes are provided that can be used to send authentication information to the DOIP server.

1) PasswordAuthenticationInfo, which sends a username and password.
2) PrivateKeyAuthentictionInfo, which given a private key will generate and send a JWT (RFC 7519).
3) TokenAuthenticationInfo, which given a (any) token will send it with the request.

Basic Operations

Methods are provided that support the 7 basic operations.

Operation	**Method name**
0.DOIP/Op.Hello	hello
0.DOIP/Op.Create	create
0.DOIP/Op.Retrieve	retrieve, retrieveElement
0.DOIP/Op.Update	update
0.DOIP/Op.Delete	delete

0.DOIP/Op.search	search, searchIds
0.DOIP/Op.ListOperations	listOperations

DigitalObject

DigitalObject is a Java class that represents the structure of a DOIP Digital Object. It contains the id, type, attributes and elements. The create, update, retrieve and search methods make use of DigitalObject in their arguments and return types.

Search

Two search methods are provided, one that returns DigitalObject instances, and one that only returns the identifiers of the objects that match the query: search() and searchIds() respectively. Both return an object of type SearchResults which implements Iterable and can also produce a Stream.

```
QueryParams queryParams = new QueryParams(0, 10);
String query = "type:Document";
try (SearchResults<DigitalObject> results
   = client.search("35.TEST/DOIPServer", query, queryParams, authInfo)) {
   for (DigitalObject result : results) {
      System.out.println(result.id + ": " + result.type);
   }
}
```

QueryParams is used in this example to specify the server to return 10 results from offset 0.

SearchResults is AutoCloseable, and so can be used in a try-with-resources statement. Without a try-with-resources statement, SearchResults must be explicitly closed at the end of results processing in order to release the connection back to the pool. SearchResults is also Iterable, and so can be used in a for-in loop.

performOperation

All of the basic operations ultimately end up being sent through the method performOperation(). This method is made public (in a Java sense) such that you can send extended operations.

The performOperation method takes an InDoipMessage and returns a DoipClientResponse. These classes and their use are described in the Javadoc.