SDK User Guide

1. Overview

The fling project provides an OpenApi JSON interface for various internal SOAP APIs, conceived via a proxy, responsible for translating JSON based requests to their SOAP equivalents. Different ways of communicating with the proxy emerged, all delivered in the following SDKs - scripting-tools, vapi-tools and openapi-tools. The current guide aims to familiarize the user with the listed SDKs.

2. vapi-tools

The vapi-tools SDK contains VMODL2 files describing the internal SOAP components. Via these files vAPI bindings are generated, which are also apparent in the delivery. These bindings are utilized by the vAPI REST-native client, in order to communicate with the proxy. They are of the following structure - models, which encompass the data transmitted on the wire, and API wrappers, which contain the operations described in a particular API.
For each operation, which has body parameters, there is a parameters wrapper model, encapsulating all models, required by the operation, in a single object.
Both models and API wrappers are logically grouped in packages and are publicly available to the SDK consumer. API wrappers are essentially stubs, which are instantiated via the REST-native's client stub factory:

```
import com.vmware.vapi.client.ApiClient;
import com.vmware.vapi.client.ApiClients;
        ...
ApiClient apiClient = ApiClients.newRestClient(proxyAddress);
        ServiceInstance serviceInstanceSvc = apiClient.createStub(ServiceInstance.class);
```

All operations contain a path parameter used for unique resource identification, namely, the managed object identifier or `moId`. To invoke an operation a `moId` needs to be always supplied and if the operation has additional parameters, the associated parameters wrapper object needs to be included:

```
SessionManager sessionManagerSvc = apiClient.createStub(SessionManager.class);
LoginParams loginParams = new LoginParams();
loginParams.setUserName(vcUsername);
loginParams.setPassword(vcPassword.toCharArray());
sessionManagerSvc.login("SessionManager", loginParams);
```

For information regarding operations or models, refer to the javadoc.

The REST-native client is stateful and manages session out of the box, thus,

once logged in with a particular client, all consecutive requests need to originate from the same client in order to be authenticated.

Problematic is the lack of support for inheritance in the vAPI runtime and bindings, because the internal SOAP components allow it. In order to represent accurately the hierarchical structure within the VMODL2 APIs composition is adopted to mimic the inheritance - meaning all subtypes @Include in themselves their parents. Since this is not real inheritance, the operations which take an abstract or base type parameter cannot acquire a particular subtype. In order to pass a subtype, one must `_convertTo` the parent (the following example is apparent in the sample):

```
CreateVirtualDiskParams createVdParams = new CreateVirtualDiskParams();
FileBackedVirtualDiskSpec fileBackedSpec = new FileBackedVirtualDiskSpec();
fileBackedSpec.setAdapterType("busLogic");
fileBackedSpec.setCapacityKb(4096);
fileBackedSpec.setDiskType("thin");
createVdParams.setDatacenter(datacenterMoRef);
createVdParams.setName(vdFolderName + "myvmdisc");
createVdParams.setSpec(fileBackedSpec._convertTo(VirtualDiskSpec.class));
```

In the code above the FileBackedVirtualDiskSpec, a subtype of VirtualDiskSpec, is being initialized and `_convertTo` its parent so it can be set in the CreateVirtualDiskParams parameter wrapper object. The `_convertTo` opeartion relies on the common representation of all vAPI models - Structure. The missing fields in the parent type, apparent in the subtype, are preserved and regarded as `dynamicFields`.

Another issue is response processing, as again, operations might return abstract or base type objects. Discrimination between returned types is done via the 'className' discriminator apparent in each model, which is unique resource identifier. Example how a specific error is discriminated:

```
MethodFault methodFault  = hostTaskInfo.getError();
if (methodFault != null && methodFault._hasTypeNameOf(SSLVerifyFault.class)) {
   SSLVerifyFault sslFault = methodFault._convertTo(SSLVerifyFault.class);
```

The `_hasTypeNameOf` method relies on the `className` discriminator.

For more information and usage guide see the delivery's README.

3. openapi-tools

The openapi-tools SDK contains an OpenApi 3 specification generated from and

describing the internal SOAP components. From the specification client bindings are produced via the third-party tool openapi-generator (https://github.com/OpenAPITools/openapi-generator). The chosen language in the samples is java and the specific flavor is resttemplate. Along with the bindings a default preconfigured `ApiClient` is generated, which is open for extension. The client (or its extended variant) is instantiated as follows:

```
ApiClient apiClient = new ApiClient();
apiClient.setBasePath(proxyAddress);
```

The generated bindings reside in two packages - model and api, accordingly, the model package contains all classes, which are used to describe the concrete data to be sent on the wire, while the api package contains API wrapper classes used to invoke particular operations. Each operations has a path parameter used for unique resource identification, namely, the managed object identifier or `moId`. To invoke an operation a `moId` needs to be always supplied and if the operation has additional parameters, a parameters wrapper object, encompassing all models to be transmitted, needs to be included. In order to instantiate an API wrapper object, the client needs to be supplied:

```
VimSessionManagerApi sessionManagerApi = new VimSessionManagerApi(apiClient);
VimSessionManagerLoginParams vimLoginParams = new
VimSessionManagerLoginParams();
vimLoginParams.setUserName(vcUsername);
vimLoginParams.setPassword(vcPassword);
vimLoginParams.setLocale("en");
sessionManagerApi.login("SessionManager", vimLoginParams);
```

Unlike the vapi bindings, the openapi ones support inheritance and no specific request nor response handling is required. Although, there are a few problems associated with the default serialization and deserialization in resttemplate. By default all unset properties in objects sent on the wire appear as `null` and unknown properties apparent in the responses cause the application to crash. To avoid such problems the default configuration needs to be overriden. Namely, a configured MessageConverter (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html) needs to be incorporated into the resttemplate:

```
private MappingJackson2HttpMessageConverter createNullDisallowingConverter() {
MappingJackson2HttpMessageConverter converter = new
MappingJackson2HttpMessageConverter();
ObjectMapper objMapper = new ObjectMapper();
objMapper.setSerializationInclusion(Include.NON_NULL);
```

```
    objMapper.registerModule(new JavaTimeModule());
    objMapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
    converter.setObjectMapper(objMapper);
    return converter;
  }
```

which specifies non-null serialization and disables fail on unknown properties.

Another problem related to the resttemplate is the lack of session support by default. There are a couple of ways to address the problem. The most worthwhile is to add a client to the resttemplate with session handling (for example, the Apaches HttpClient). The sample code utilizes a custom request interceptor, attached to the resttemplate, responsible for acquiring and persisting the session cookies:

```
  private class SessionAwareRestTemplateInterceptor
   implements ClientHttpRequestInterceptor {
    private String session;

    @Override
    public ClientHttpResponse intercept(HttpRequest req,
                                byte[] body,
                                ClientHttpRequestExecution execution)
                                      throws IOException {
      if (session != null) {
        req.getHeaders().add(HttpHeaders.COOKIE, session);
      }
      ClientHttpResponse response = execution.execute(req, body);

      if (session == null) {
        session = response.getHeaders()
              .getFirst(HttpHeaders.SET_COOKIE);
      }
      return response;
    }
  }
```

For more information and usage guide see the delivery's README.

4. scripting-tools
The scripting-tools delivery serves solely as a showcase of the wire representation of requests and responses between the proxy and a client - in the current case postman.

For more information and usage guide see the delivery's README.