z/TPF REST Education

2023 TPF Users Group Conference
April 24-26, Dallas, TX
Ongoing TPF Education

—

Bradd Kadlecik

IBM

Agenda

- Overview
- Getting Started
- Deploying to z/TPF
- Configuration Options
- Problem Determination

Agenda

- **Overview**
- Getting Started
- Deploying to z/TPF
- Configuration Options
- Problem Determination

REST provider architecture

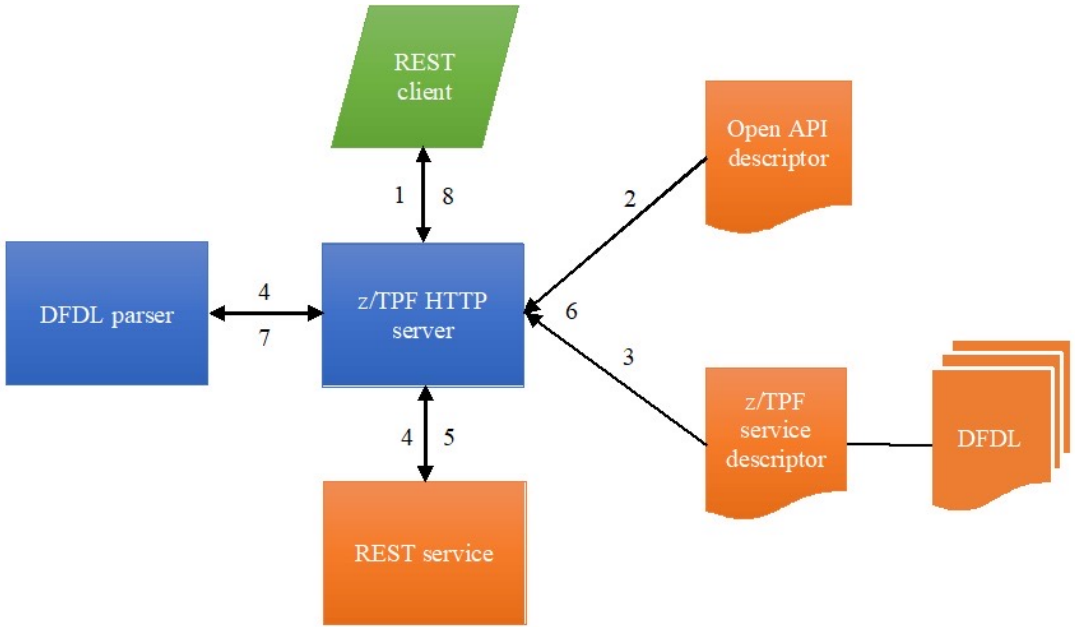
Artifacts:

OpenAPI descriptor - description of REST interface

DFDL descriptor - description of data interface

z/TPF service descriptor - configuration options

JAM descriptor (optional) - used for Java interface



REST consumer architecture

Artifacts:

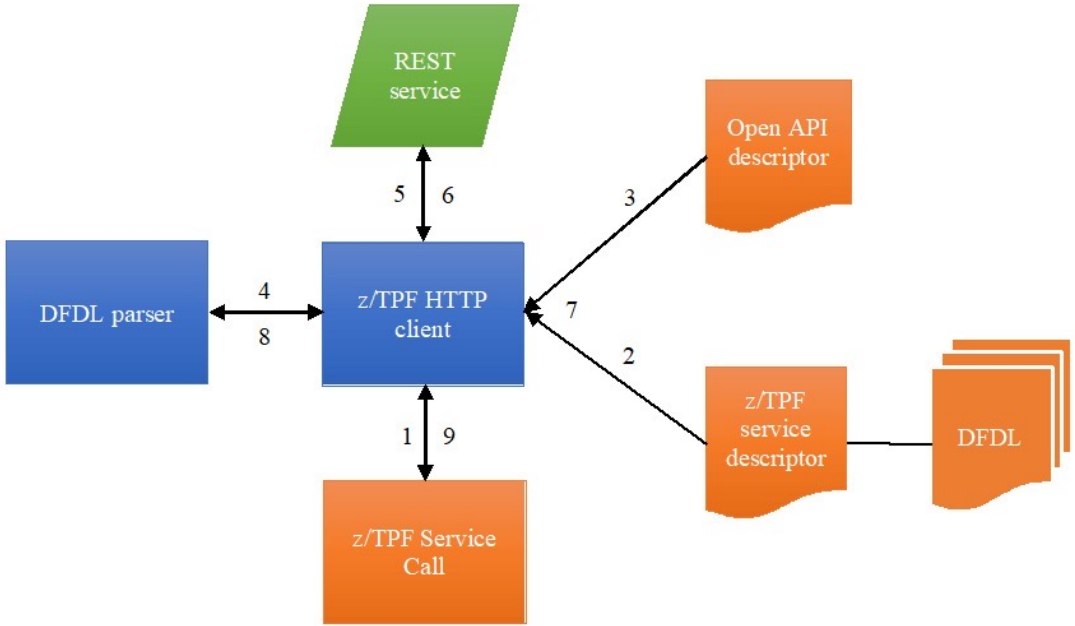
OpenAPI descriptor

DFDL descriptor

z/TPF service descriptor

JAM descriptor (optional)

Called through
tpf_srvcInvoke* APIs



Java optimized REST service interface

Artifacts:

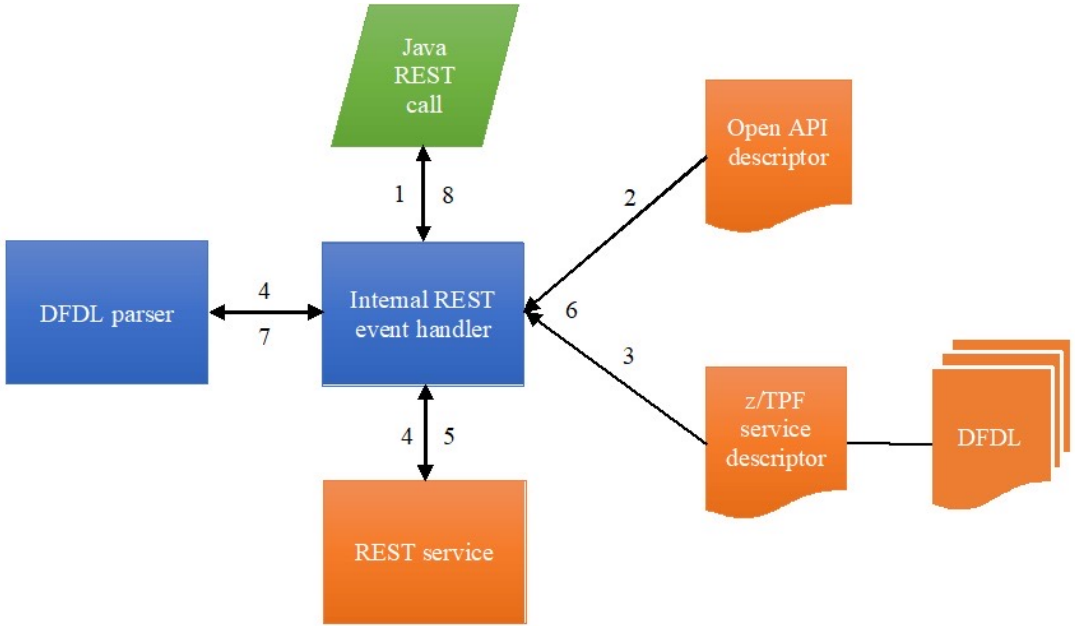
OpenAPI descriptor

DFDL descriptor

z/TPF service descriptor

“host” must be “localhost”

“basePath” used as unique key for lookup



REST name-value pairs

REST provider automatically sets the following name-value pairs:

ISrvcName – operationId (service name) of the request

ISrvcVersion – REST API version (defined by OpenAPI descriptor)

Both can be retrieved using *tpf_srvcGetInfo*.

Benefits:

Name-value pair collection

Real-time runtime metrics collection

Dump console display

OpenAPI descriptor
minimal definition

```
{
  "swagger":"2.0",                                  <- support only 2.0
  "host":"host.example.com:81",                     <- required only for REST consumer or localhost for Java to z/TPF
  "info":{
    "description":"REST services for sample package",
    "version":"1.0.0",                              <- value for ISrvcVersion name-value pair
    "title":"HTPD Services"
  },
  "basePath":"/tpf/sample",
  "paths":{
    "/sync":{                                       <- basePath + path extension = full URI
      "get":{                                       <- HTTP method
        "operationId":"htpdSyncGet",                <- value for ISrvcName name-value pair (service name)
        "responses":{
          "200":{
            "description":"normal response"
          }
        }
      } } } }
```

z/TPF service descriptor
minimal definition

```
{
  "version":1,
  "description":"REST services for sample package",
  "providerType":"HTTPProgram",                    <- defines application interface
  "provider":"QHD1",
  "timeout":2000,
  "services": [
  { "operationId":"htpdSyncGet" }                  <- service name (matches operationId in OpenAPI)
}
```

REST provider request

GET http://mytpf:81/tpf/sample/sync?parm1=90210
content-type: “application/json”
content-length: “250”
parm2: “76”
{ “prg1_input”: .... }

route by operationId (defined in z/TPF service descriptor as program QHD1)

REST provider response

tpf_srvcSendResponse

tpf_srvc_token -> associated with connection and operationId
tpf_srvc_resp.data -> data for transformation
tpf_srvc_resp.datalen
tpf_srvc_resp.status -> HTTP status code (formats can differ by status)
tpf_srvc_resp.status_reason -> HTTP status text

Status code: <status> <status_reason>
parm2: ...
{“prg1_output”: ... }

REST provider interfaces

“providerType”:”**Program**”

void <PROG>
(struct <input_struct> *request,
unsigned int length,
tpf_srvc_token token);

DFDL transforms the HTTP request into a user defined request structure.

Respond using token on same or different ECB with *tpf_srvcSendResponse.*

“providerType”:”**HTTPProgram**”

void <PROG>
(tpf_httpsvr_req *request,
tpf_httpsvr_token token);

No DFDL needed. This interface predates REST provider.

Respond using token on same or different ECB with *tpf_httpSendResponse.*

REST consumer interfaces

tpf_srvcInvoke(“operationId”, reqData, reqLen, &srvcResp, 0);

- Synchronous
- Uses DFDL to transform request data to a HTTP request and the HTTP response to response data.

tpf_srvcInvoke_ext(“operationId”, srvcReq, &srvcResp, asyncParms, 0);

- Synchronous or Asynchronous
- Can transform data using DFDL or interact directly with HTTP request/response data.

Agenda

- Overview
- **Getting Started**
- Deploying to z/TPF
- Configuration Options
- Problem Determination

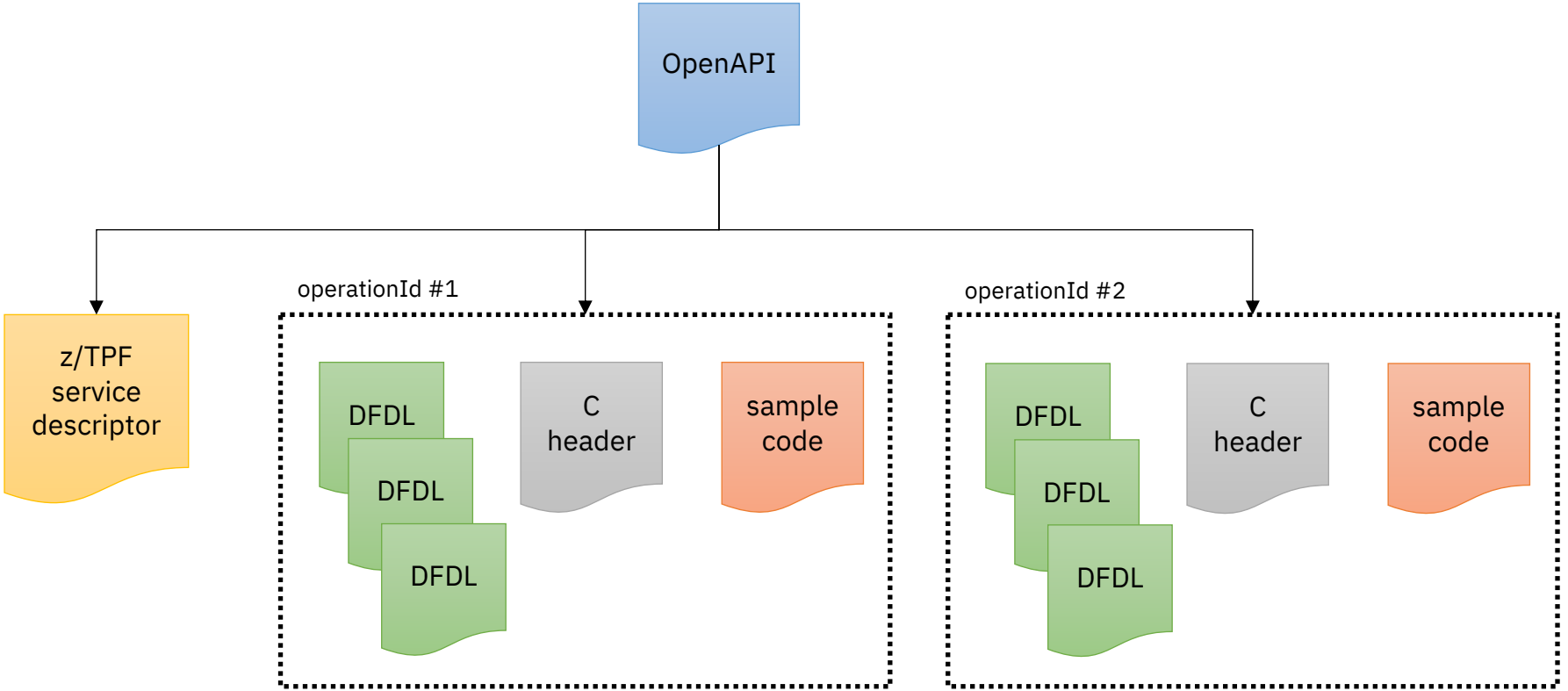
Getting started
Creating REST artifacts

Two methods

1. Begin with the OpenAPI descriptor.
   - Generate DFDL schemas, z/TPF service descriptor, and C structures by using tpfrestgen.
2. Begin with C structures.
   - Generate DFDL using tpfdfdlgen (or maketpf <program> dfdl).
   - Use tooling to create the OpenAPI descriptor from the DFDL (XML schema needs to be converted to JSON schema for the OpenAPI descriptor).
   - Create the z/TPF service descriptor.

OpenAPI code generation utility (tpfrestgen)

Shipped with z/TPF on Linux on IBM Z



tpfrestgen generated C structures
OAS (**O**pen**A**PI **S**pecification) format

“DFDLFormat”:”OAS” <- defined in z/TPF service descriptor

```
typedef struct {
      struct {
      } header;       <- contains all header parameters
      struct {
      } path;         <- contains all path parameters
      struct {
      } query;        <- contains all query parameters
      struct {
      } body;         <- contains all body parameters (order matches JSON schema)
} <operationId><request/response>_t;        <- root element
```

REST non-generated C structures

“DFDLFormat”:”none” <- default

- Order must match JSON schema in OpenAPI descriptor.
- Root element is an optional object definition in the OpenAPI JSON schema.
- Path, query, and header parameter names must be unique (cannot match a name in the body or elsewhere otherwise must use OAS format).
- Any fields not added as parameters to the OpenAPI descriptor are ignored (hidden).

tpfrestgen generated C structures
strings

Fixed length string

OpenAPI JSON schema:

```
“name”:{
      “type”:”string”,
      “maxLength”:20
}
```

C structure definition:

```
char name[20];
```

Variable length string

OpenAPI JSON schema:

```
“name”:{
      “type”:”string”
}
```

C structure definition:

```
char *name;
```

tpfrestgen generated C structures
array of strings

Fixed length array

```
“name”:{
      “type”:”array”,
      “maxItems”:5,
      “items”:{
            “type”:”string”,
            “maxLength”:20
      }
}
```

```
char name[5][20];
```

Variable length array

```
“name”:{
      “type”:”array”,
      “items”:{
            “type”:”string”,
            “maxLength”:20
      }
}
```

```
u_int32_t nameItemCount;
char (*name)[20];
```

tpfrestgen generated C structures
types of string arrays

u_int32_t varStringArrayItemCount;
char **varStringArray; /* variable size array of variable length string */

u_int32_t stringArray1ItemCount;
char *stringArray1[5]; /* fixed size array of variable length string */

u_int32_t stringArray2ItemCount;
char (*stringArray2)[20]; /* variable size array of fixed length string */

u_int32_t stringArray3ItemCount;
char stringArray3[5][20]; /* fixed size array of fixed length string */

tpfrestgen generated C structures
variable size array of objects

```
u_int32_t varComplexArrayItemCount;
struct {
    char *customer;
    int32_t order;
    char *dest;
} *varComplexArray;

response.body.varComplexArray = calloc(varComplexArrayItemCount,
                                       sizeof(*varComplexArray));

for (int i = 0; I < response.body.varComplexArrayItemCount; i++) {
    response.body.varComplexArray[i].customer =
}
```

Generating array counts in DFDL
ItemCount

int stringArray**ItemCount**; <- matching field name with suffix “ItemCount”
char **stringArray;

```
<xs:element name="stringArrayItemCount" type="xs:int" dfdl:lengthKind="explicit"
dfdl:length="4" dfdl:lengthUnits="bytes" dfdl:outputValueCalc="{fn:count(../stringArray)}"
default="0"/>
    <xs:sequence tddt:indirectKind="pointer" tddt:indirectLength="8">
        <xs:element name="stringArray" type="xs:string" tddt:indirectKind="pointer"
tddt:indirectLength="8" dfdl:lengthKind="delimited" dfdl:lengthUnits="bytes"
dfdl:terminator="%NUL;" dfdl:textTrimKind="none" minOccurs="0"
maxOccurs="unbounded" dfdl:occursCountKind="expression"
dfdl:occursCount="{../stringArrayItemCount}" default=""/>
    </xs:sequence>
</xs:element>
```

Agenda

- Overview
- Getting Started
- **Deploying to z/TPF**
- Configuration Options
- Problem Determination

Deployment options

All descriptor files must first be loaded to /sys/tpf_pbfiles/tpf-fdes on z/TPF.

1. By operator commands
   ZMDES DEPLOY FILE-<OpenAPI filename>
   ZHTPS ADD Server-<server> File-<OpenAPI filename>
2. By REST service
   POST /tpf/srvcmgmt/service

Note: Using url_program_map.conf to update the HTTP servers is **not** recommended as it adds the service for all servers (SSL and non-SSL).

Descriptor deployment

DFDL – automatically and permanently deployed
- key – filename, type – DFDL (ZMDES LOC)
- system heap owner name – IDFDL (ZSTAT)

z/TPF service descriptor – automatically deployed
- key – filename, type – SRVC (ZMDES LOC)
- system heap owner name – ISERVICE (ZSTAT)

OpenAPI descriptor – manually deployed (must use ZMDES DEPLOY)
- key – filename, type – SWAGGER (ZMDES LOC)
- system heap owner name – IOPENAPI (ZSTAT)

Monitoring a REST service
Usage statistics

```
User:   ZSRVC STATS NAME-itpftest*

System: SRVC0004I 13.17.33 REST SERVICES STATISTICS DISPLAY
NAME             VERSION   ACT    HW   MAX  REJECT TIMEOUT IFERROR SCERROR
itpftest2          1.0.0    44    50    50     164      17       0       0
itpftest3          1.0.0    23    23   100       0       1       0       5
itpftest1          1.0.0     0    10     0       0       2       0       0
itpftest           1.0.0     0     0    50       0       0       0       0
END OF DISPLAY+
```

ACT – Active requests
HW – highwater count (sort key)
MAX – max concurrent allowed

REJECT – number of requests rejected by throttling
TIMEOUT – number of requests timed out
IFERROR – number of interface errors encountered
SCERROR – number of service errors (status returned >= 300)

Monitoring a REST service
Runtime metrics collection

- Message rate
- CPU usage
- Existence time
- Find/Files
- z/TPFDF usage
- ECB heap usage

Removing a REST service

ZMDES UNDEPLOY FILE-<OpenAPI filename>

ZHTPS REMOVE S-<server> F-<OpenAPI filename>

Descriptor files are removed using the following loader input control file statements:

@FILE

@@DELETE <filepath>

Agenda

- Overview
- Getting Started
- Deploying to z/TPF
- **Configuration Options**
- Problem Determination

REST logging user exits

REST provider uses the HTTP server which contains

HTTP server request/reply for logging user exit:

UHSR (UHSR_input *input);

REST consumer uses the enhanced HTTP client which contains

Enhanced HTTP client request/reply for logging user exit:

UHCR (UHCR_input *input);

Enabling HTTP server compression

The z/TPF HTTP server can compress HTTP responses when requested by a client that uses the “Accept-Encoding” header.

To enable compression, update the following in the HTTP server configuration file:

- COMP – prioritized list of supported compression options (deflate, gzip).
- COMPSIZEHW – minimum size of the response body to perform compression when hardware is available.
- COMPSIZESW – minimum size of response body to perform compression when hardware is not available.

Using high speed connector with REST consumer

The host is configured for REST consumer through the “host” property in the OpenAPI descriptor.

```
{
    “swagger” : “2.0”,
    “host” : “sampleProvider”,
```

The host can be set to an aliasHostname of an endpoint group descriptor.

```
<aliasHostname>sampleProvider</aliasHostName>
```

z/TPF service descriptor options
“exclude” : “all”

DFDL elements that are not required can be excluded if there is a default value defined somewhere:

1. XSD default - only 0 or empty string allowed
   <xs:element name="userToken" type="xs:string" dfdl:lengthKind="explicit" dfdl:length="7" dfdl:trailingSkip="1" dfdl:lengthUnits="bytes" default="" />
2. DFDL nilValue – any literal value, preferably one that can’t be a logical value
   <xs:element name="userToken" type="xs:string" dfdl:lengthKind="explicit" dfdl:length="7" dfdl:trailingSkip="1" dfdl:lengthUnits="bytes" nillable="true" dfdl:useNilForDefault="yes" dfdl:nilKind="literalCharacter" dfdl:nilValue="%NUL;" />
3. OpenAPI default – any logical value

```
"userToken" : {
  "type" : "string",
  "default" : "none",
}
```

z/TPF service descriptor options
“exclude” : “all”

“exclude” : “none”

```
{
 “apar”:”PJ45427”,
 “reviewers”:[
    {“name”:”John Smith”,
     “date”:”2018-10-11”},
    {“name”:””,
     “date”:””}
 ],
 “coreqs”:false,
 “miginfo”:true
}
```

“exclude” : “all”

```
{
 “apar”:”PJ45427”,
 “reviewers”:[
    {“name”:”John Smith”,
     “date”:”2018-10-11”}
 ],
 “miginfo”:true
}
```

z/TPF service descriptor options
REST throttling

By maximum concurrent requests:

“maxRequests” – number requests allowed

“maxRequestsError” – status code to return

“maxRequestsWarningInterval” – warning message frequency

By resource priority class:

“priorityClass” – LODIC resource priority class

“priorityError” – status code to return

“priorityWarningInterval” – warning message frequency

Supporting multiple API versions

OpenAPI descriptors can use 2 different file extensions:

- .swagger.json -> operationIds are subsystem unique
- .openapi.json -> operationIds are document unique

There are 2 versions of z/TPF service descriptors:

- “version”:1 -> used to define z/TPF options for REST APIs in a .swagger.json
- “version”:2 -> used to define z/TPF options for REST APIs in a .openapi.json

The version designation should be part of the OpenAPI Info Object and basePath.

- “version” : “1.0.0”
- “basePath” : “/pgmmgt/v1”

Agenda

- Overview
- Getting Started
- Deploying to z/TPF
- Configuration Options
- **Problem Determination**

REST artifact validation

1. DFDL schema files can be validated on Linux on IBM Z using *tpfdatamap verify*.
2. OpenAPI and z/TPF service descriptors are validated during *ZMDES DEPLOY FILE-<OpenAPI filename>*. This command can always be rerun for validation purposes even if already deployed.

REST deployment validation

ZSRVC Display validates all related descriptors are deployed.

```
User:    ZSRVC DISPLAY NAME-oasComplex*
System:  SRVC0006I 18.37.40 REST SERVICE PROPERTIES AND ARTIFACTS DISPLAY
THE FOLLOWING SERVICES HAVE AN UNDEPLOYED OPENAPI DESCRIPTOR
  oasComplexReq1          oasComplexReq2          oasComplexReq3
  /oasServices/v2/oasComplexReq1 _
  /oasServices/v2/oasComplexReq2
  /oasServices/v2/oasComplexReq3
END OF DISPLAY+
```

Status error 404 – Not Found
Verify by REST service

GET /tpf/srvcmgmt/service

"name": <OpenAPI Info Object title>
"version": <OpenAPI Info Object version>
"description": <OpenAPI Info Object description>
"available": <true if deployed to this server>
"restricted": <true if deployed by server, not in url_program_map.conf>
"deployed": <true if OpenAPI deployed>
"file_name": <OpenAPI filename>
"documentation": <URL to retrieve OpenAPI descriptor>

Status error 404 – Not Found
Verify by operator commands

1. Verify OpenAPI descriptor is deployed.

   ZMDES DISPLAY FILE-<OpenAPI filename> or

   ZSRVC Display Name-<service name>

2. Verify REST service is deployed to server.

   ZHTPS DISplay Server-<server> URL

Other status errors

405 (Method Not Allowed) – Verify method is defined in OpenAPI descriptor.

ZSRVC Display Name-<operationId> (displays method, URI)

406 (Not Acceptable) – Verify HTTP request format

Check status reason for more information (capture with logging user exit)

500 (Internal Server Error) – Multiple scenarios, possibly bad response data

Check status reason for more information (capture with logging user exit)

503 (Service Unavailable) – Verify z/TPF service descriptor is deployed

ZMDES DEPLOY FILE-<OpenAPI filename> (verifies all descriptors)

No data received or missing data

Data is created using default values when the request doesn’t match the OpenAPI or DFDL definitions.

No element/property is required, and additional properties are ignored; therefore, **any** type of request will be accepted!

Typically, there are a few elements that are always expected in a request (and should be indicated as required).

Updating the OpenAPI descriptor to indicate the parameters or properties that are required will allow requests that don’t match definitions to get rejected with information on what was missing (or doesn’t match the definition).

Setting required elements in OpenAPI

Header, Query, Path parameters:

```
“parameters” : [
      {
            “in” : ”header”,
            “name” : ”Authorization”,
            “required” : true,
            “type” : ”string”
      }
]
```

Body properties:

```
“customer” : {
      “type” : “object”,
      “properties” : {
            “name” : {
                  “type” : “string”
            }
      },
      “required” : [“name”]
}
```

No data received or missing data

Another reason for missing data is when the JSON, OpenAPI JSON schema, and DFDL schema do not all line up.

1. Check the z/TPF service descriptor “unordered” value. If there is no “unordered” property or it is set to *false*, then any element that appears out of order in the JSON will not be found and a default value will be used instead.
2. Verify that the JSON schema properties in the OpenAPI descriptor for the fields with missing values appear in the same order as the fields in your structure.

Same order C struct and JSON schema

C struct:

```
struct {
        char *name;
        int id;
} customer;
```

OpenAPI JSON schema:

```
“customer” : {
        “type” : “object”,
        “properties” : {
                “name” : {
                        “type” : “string”
                },
                “id” : {
                        “type” : “integer”
                }
        }
}
```

Capturing REST requests/replies

It can be valuable to see what DFDL is receiving or creating when problems occur.

The logging user exits are a great place to capture how things look or find what status reason is being passed back.

Data is in ASCII but can be easily viewed using ZFILE dd if=<filename> conv=fromlinux.

```
void UHSR (UHSR_input *input) {

 char *filename;
 int fd;

 if (input->type)
    filename = "/tmp/request.txt";
 else
    filename = "/tmp/response.txt";

 fd = open(filename, O_WRONLY |
      O_CREAT, S_IRUSR | S_IROTH);
write(fd, input->http_msg_format,
      input->http_msg_len);
close(fd);
```

Helpful links

- IBM Documentation
  https://www.ibm.com/docs/en/ztpf/2023?topic=support-rest
- OpenAPI Specification
  https://swagger.io/specification/v2/
- REST versioning
  https://public.dhe.ibm.com/software/htp/tpf/tpfug/tgs22/TPFUG_2022_WEBS_REST.pdf
- REST challenge on the learning platform
  https://ibmzxplore.influitive.com/channels/10

Thank you