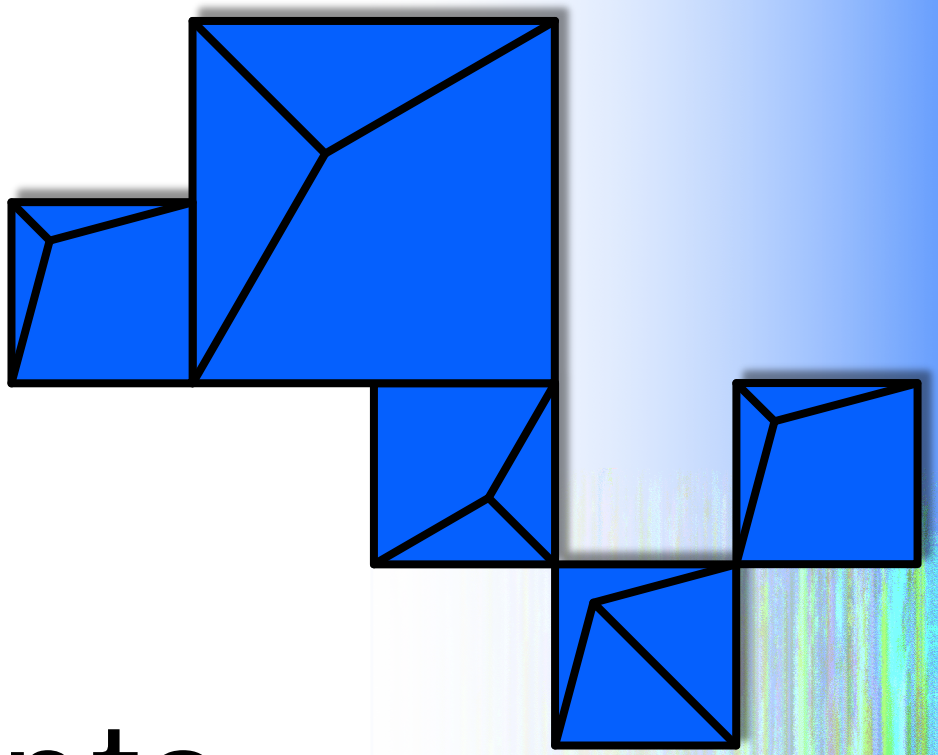


REST Enhancements

Bradd Kadlecik

Disclaimer

Any reference to future plans are for planning purposes only. IBM reserves the right to change those plans at its discretion. Any reliance on such a disclosure is solely at your own risk. IBM makes no commitment to provide additional information in the future.

Agenda

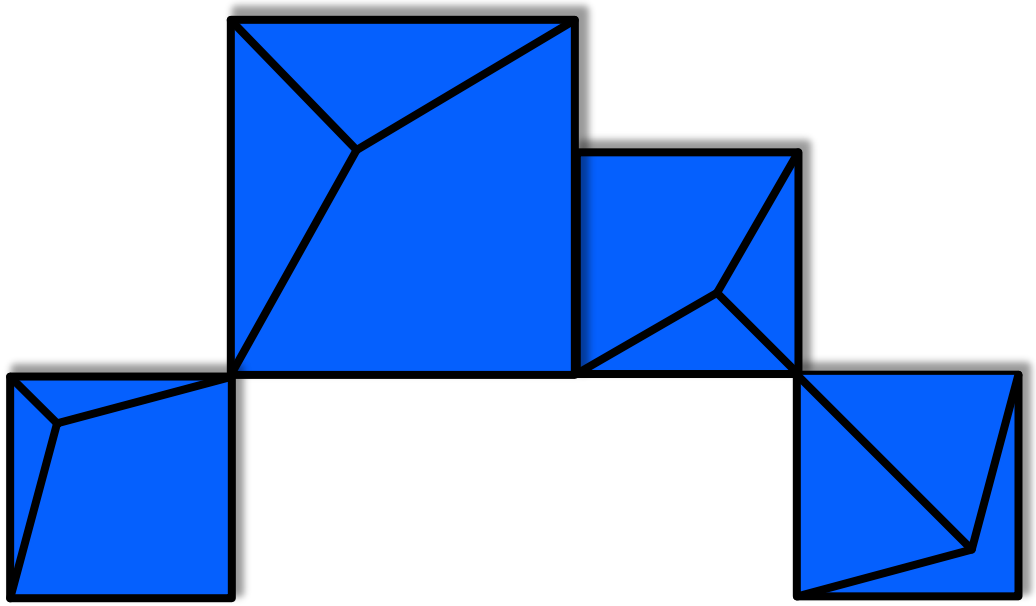
- Background
- New features
- Supporting multiple API versions
- Generating REST artifacts
- Deploying REST services
- Conclusion

Background

- REST provider and REST consumer seek to provide a simplified way of creating RESTful interfaces based on OpenAPI and DFDL.

- OpenAPI is a standardized description for RESTful APIs while DFDL is a standardized description for data.

- REST provider and REST consumer transform data to/from HTTP requests and responses.



New features (PJ45579 – June 2019)

Problem Statement

Certain data in REST messages is not available for applications to process using REST provider and REST consumer.

Pain Points

- Some REST APIs pass variable data in the URI but REST consumer and provider don’t support path parameters.
- Some characters allowed in JSON are not allowed in DFDL element names.
- XML may contain attributes that can’t be defined in DFDL.
- A REST API may have multiple responses based on status code but only 1 DFDL can be defined.

Value Statement

REST consumer and provider can support more of what can be defined through the OpenAPI specification.

Technical Details

- Parameters in OpenAPI can be in the body, header, query, or path.
- An XML object name can assign a DFDL/XML name to a JSON name to allow for non-compatible characters.
- An XML object can be used to specify which elements are XML attributes.
- The z/TPF service descriptor supports an array of response objects. There can be a default response object as well as response objects for specific status codes.

Example: path parameters

PUT http://mytpf:81/tpf/service/loadset/appltest

OpenAPI definition:

```
“/loadset/{lsname}”:{
  “put”:{
    “operationId”:”tpfModLset”,
    “parameters”:[
    {
      “name”:”lsname”,
      “in”:”path”,
      “required”:true,
      “type”:”string”
    }
```

Example: non-compatible JSON name

JSON:

```
{
   “stdhd”:{
      “@stdbid”:”BD”,
      “stdchk”:1,
      “stdctl”:0,
      “stdpgm”:”ABCD”,
      “stdfch”:0,
      “stdbch”:0
   }
}
```

OpenAPI definition:

```
“stdhd”:{
   “type”:”object”,
   “properties”:{
      “@stdbid”:{
         “type”:”string”,
         “xml”:{
            “name”:”stdbid”
         }
      },
```

Example: XML attribute

XML:

```
<stdhd stdbid="BD">
  <stdchk>1</stdchk>
  <stdctl>0</stdctl>
  <stdpgm>ABCD</stdpgm>
  <stdfch>0</stdfch>
  <stdbch>0</stdbch>
</stdhd>
```

OpenAPI definition:

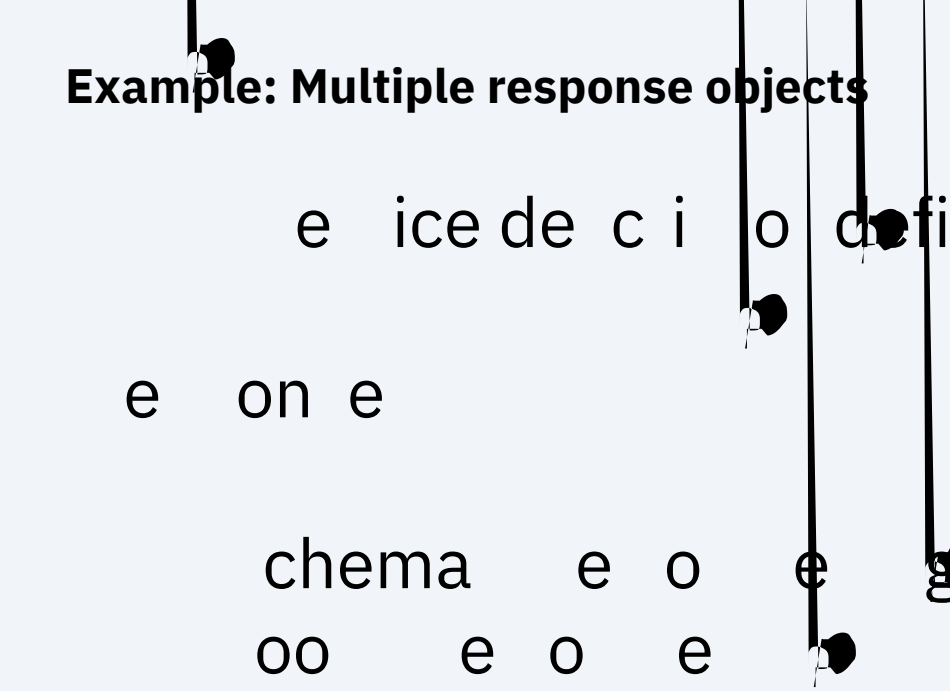
```
"stdhd":{
   "type":"object",
   "properties":{
      "stdbid":{
         "type":"string",
         "xml":{
            "attribute":true
         }
      },
```

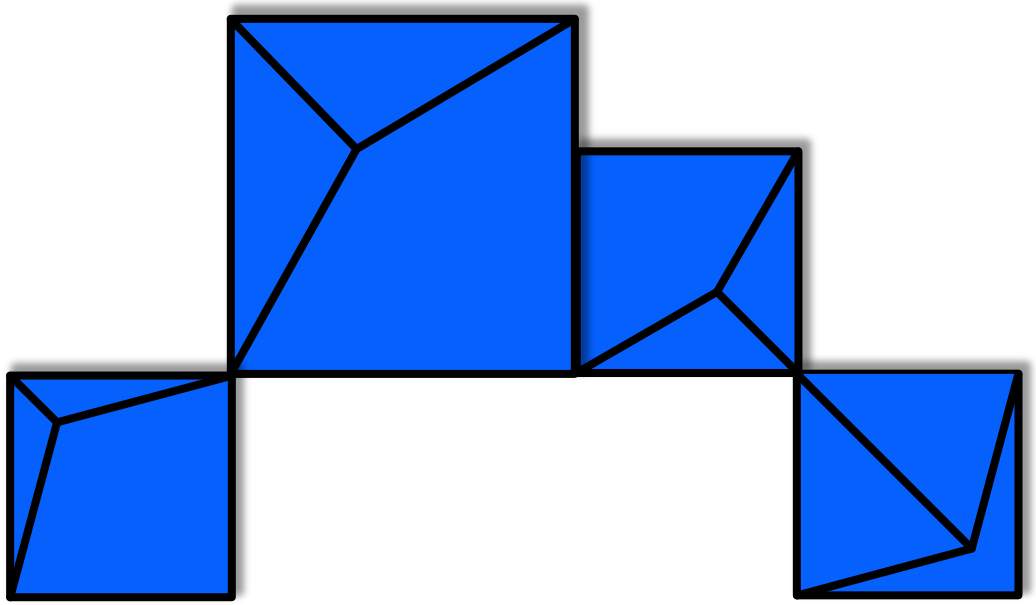
Example: Multiple response objects

OpenAPI definition:

```
"default": {
   "description": Default error response"
   "schema": {
      "$ref": "#/definitions/error_response"
   }
},
"200": {
   "description": "Successful response",
   "schema": {
      "$ref": "#/definitions/service_response"
   }
}
```

Example: Multiple response objects





Supporting multiple API versions (PJ45968 – Jan 2020)

Background

- There are several strategies for handling migration, fallback, and coexistence of REST APIs.

- One strategy is to have a REST API always be backward compatible which requires the server to handle all issues related to migration, fallback, and coexistence.

- Another strategy is to have multiple versions of REST APIs which requires the client to handle all issues related to migration and fallback while the server manages coexistence.

Problem Statement

For REST provider to manage multiple versions of REST APIs, all operationIds need to be changed every time another version of the OpenAPI descriptor is loaded.

Pain Points

- z/TPF requires operationIds to be subsystem unique while the OpenAPI specification requires them to only be document unique.

- The OpenAPI descriptor used by z/TPF cannot be the same as that used by the rest of the enterprise since the operationIds must be different for each new version.

- Changing every operationId and creating associated z/TPF service descriptors any time just one of the REST APIs in an OpenAPI descriptor is changed is tedious.

Value Statement

Multiple versions of REST APIs can coexist in z/TPF with the same operationId.

Technical Details

- 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 a REST API in .swagger.json
  - “version”:2 -> used to define a REST API in .openapi.json

- REST provider sets 2 name-value pairs:
  - ISrvcName -> the operationId
  - ISrvcVersion -> the “version” value in the OpenAPI Info Object.

Technical Details

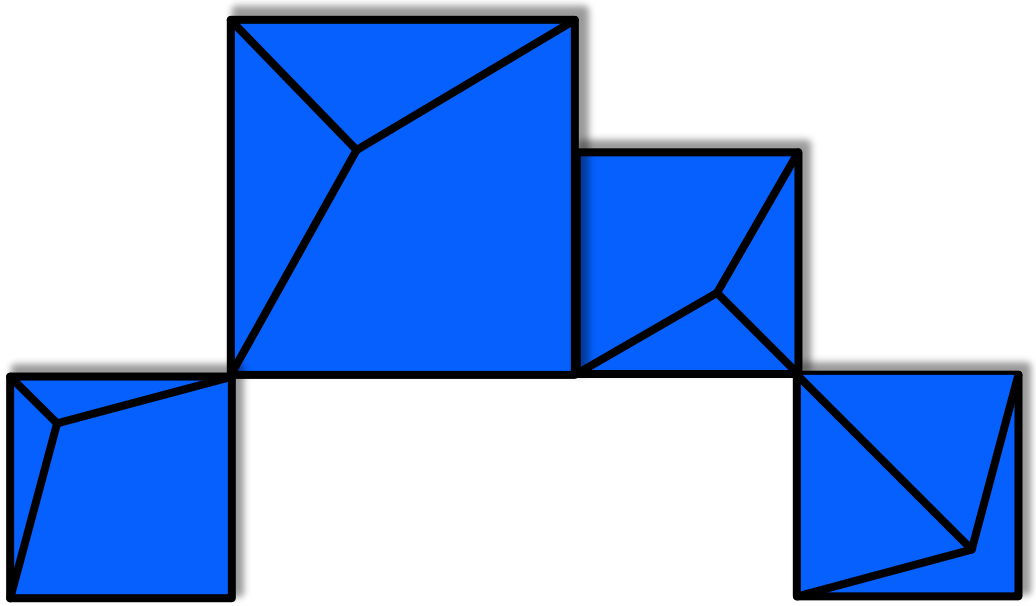
- The “basePath” in an OpenAPI descriptor must still be subsystem unique.
- An OpenAPI descriptor describes all REST APIs that use that basePath with path extensions.
- The version designation should be part of the basePath.

  ex. /pgmmgt/v1/loadset
- Name-value pair uses the “version” value in the OpenAPI Info Object to understand the version.

  “info”:{

  “version”: “1.0.0”
- New function tpf_srvcGetInfo can retrieve the following:

  operationId, basePath, info object version



Generating REST artifacts (PJ45897 – June 2020)

Problem Statement

It’s easy to create REST services on other platforms from an OpenAPI document but difficult on z/TPF.

Pain Points

- There’s no tool for generating C artifacts from an OpenAPI descriptor making this a **long** and **difficult** process.

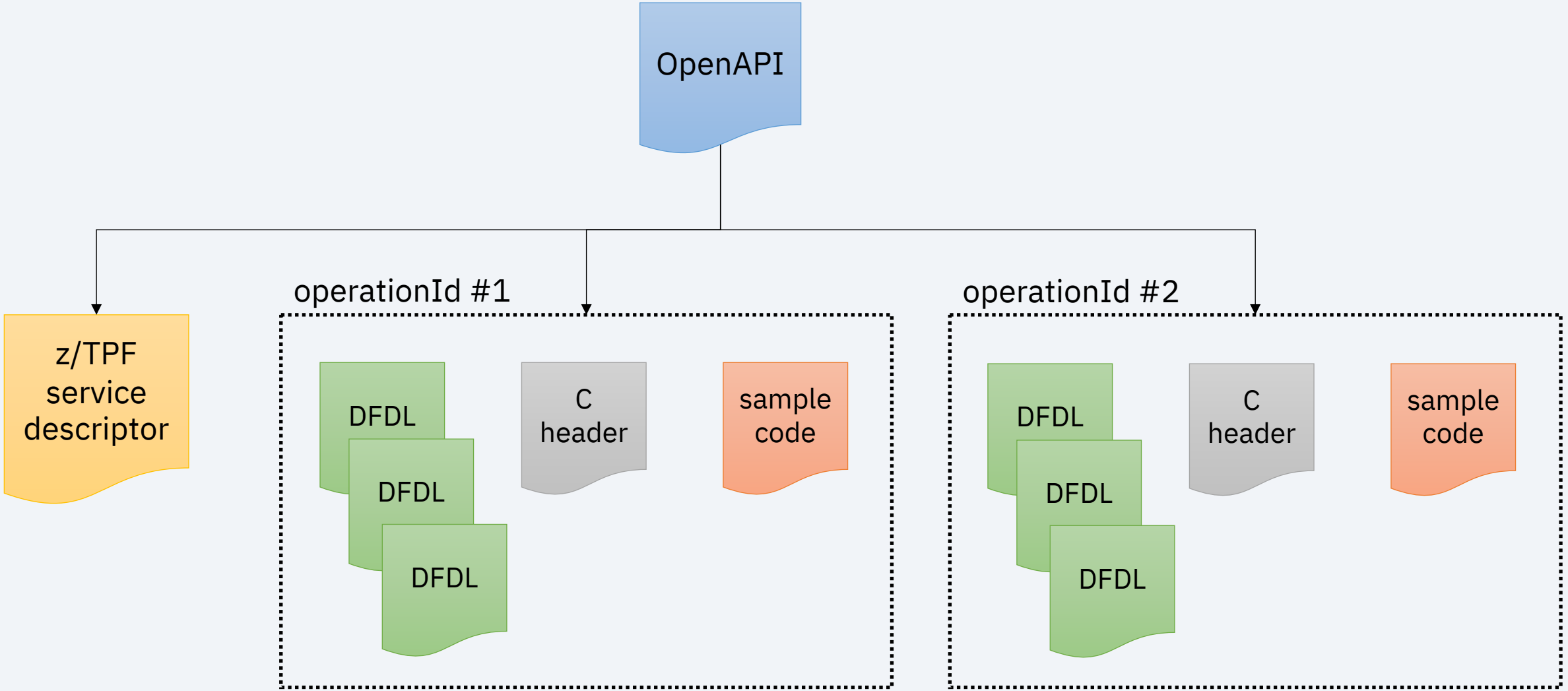
- DFDL creates flattened representations that are difficult to navigate with **variable length items**.

Value Statement

An application programmer can generate z/TPF REST artifacts for each service in an OpenAPI descriptor that are ready to deploy.

OpenAPI code generation utility

tpfrestgen <OpenAPI file> -o <output dir>



Variable size arrays

Array of objects:

```
u  int      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 =
}
```

Variable size arrays

Array of strings:

```
u  int      t varStringArrayItemCount
char     varStringArray

response.body. varStringArray = calloc(varStringArrayItemCount,
                                                    sizeof(* varStringArray));
for (int i = 0; I < response.body. varStringArrayItemCount; i++) {
      response.body. varStringArray[i] =
}
```

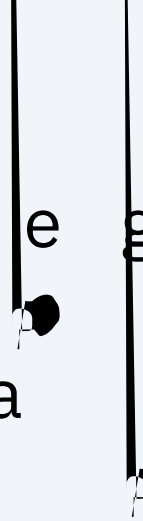
Types of string arrays

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

u_int32_t stringArray1ItemCount;
char   stringArray               /* fixed size array of variable length string */

u_int32_t stringArray2ItemCount;
char    stringArray      0       /* variable size array of fixed length string */

u_int32_t stringArray3ItemCount;
char stringArray         0       /* fixed size array of fixed length string */
```

Fixed v Variable

tpfrestgen **-mi 64**

```
char *stringArray1[5];

“stringArray1”: {
   “type”: “array”,
   “items”: {
      “type”: “string”
   },
   “maxItems”: 5
}
```



f e gen **-ml 256**

cha ing a

ing a

e a a

i em

e ing

maxLength

ma em

Generated data layout

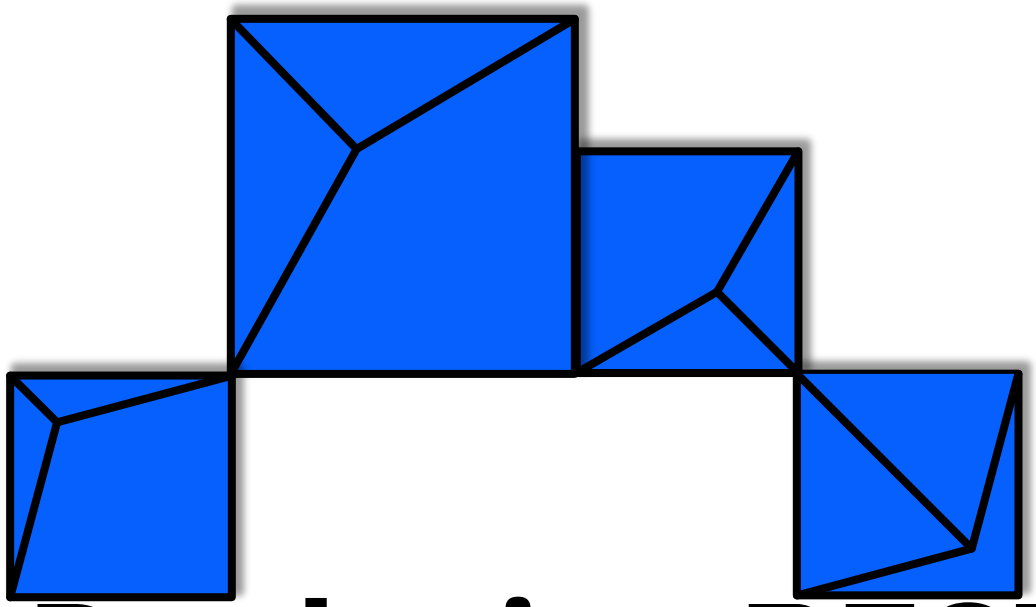
```
typedef struct {
  struct {
    char *parm1;
  } header;
  struct {
    char *parm2;
  } query;
  struct {
    u_int32_t varComplexArrayItemCount;
    struct {
      char *customer;
      int32_t order;
      char *dest;
    } *varComplexArray;
  } body;
} operationId1Request_t;
```

z/TPF service descriptor

```
{
  "version": 1,
  "unordered": true,
  "timeout": 5000,
  "exclude": "all",
  "DFDLFormat": "OAS",
  "services": [
     ...
  ]
}
```

Technical Details

- Prereqs:
  - PJ45953 (online DFDL update)
  - PJ45994 (tpfdfdlgen update)
- The tpfrestgen utility is shipped with z/TPF in tpftools directory
- Same environment requirements to run as maketpf



Deploying REST services (PJ46043 – May 2020, PJ46010 – August 2020)

Problem Statement

Creating REST services on z/TPF is somewhat complicated and error prone.

Pain Points

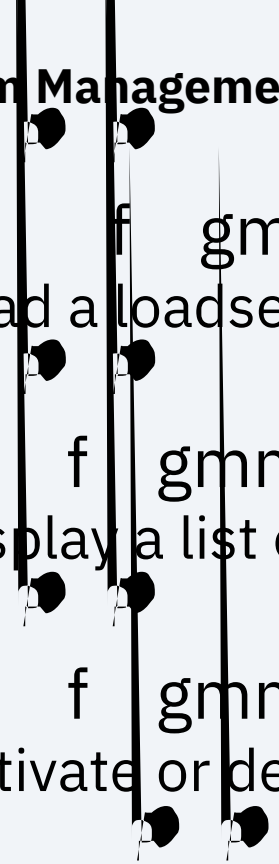
- Manually updating the URL program mapping file is both tedious and seemingly unnecessary.

- Multiple steps can be difficult to remember and cause a step to be forgotten.

Value Statement

An application developer can deploy a REST service to a z/TPF test system with a single action.

Program Management REST services – PJ46043



f gmmgm load e

Load a loadset, specifying either filename or DD name (ZOLDR LOAD)

f gmmgm load e

Display a list of all loadsets (ZOLDR DISP ALL)

f gmmgm load e *{loadset}*

Activate or deactivate a loadset (ZOLDR ACT|DEAC)

f gmmgm load e *{loadset}*

Display status and contents of a loadset (ZOLDR DISP L)

f gmmgm load e *{loadset}*

Delete a loadset (ZOLDR DEL)

Service Management REST services – PJ46010

POST /tpf/srvcmgmt/service

- Add a REST service to a HTTP server (ZHTPS ADD)
- Also deploys service if undeployed (ZMDES DEPLOY)

GET /tpf/srvcmgmt/service

- Display a list of REST services and status

DELETE /tpf/srvcmgmt/service

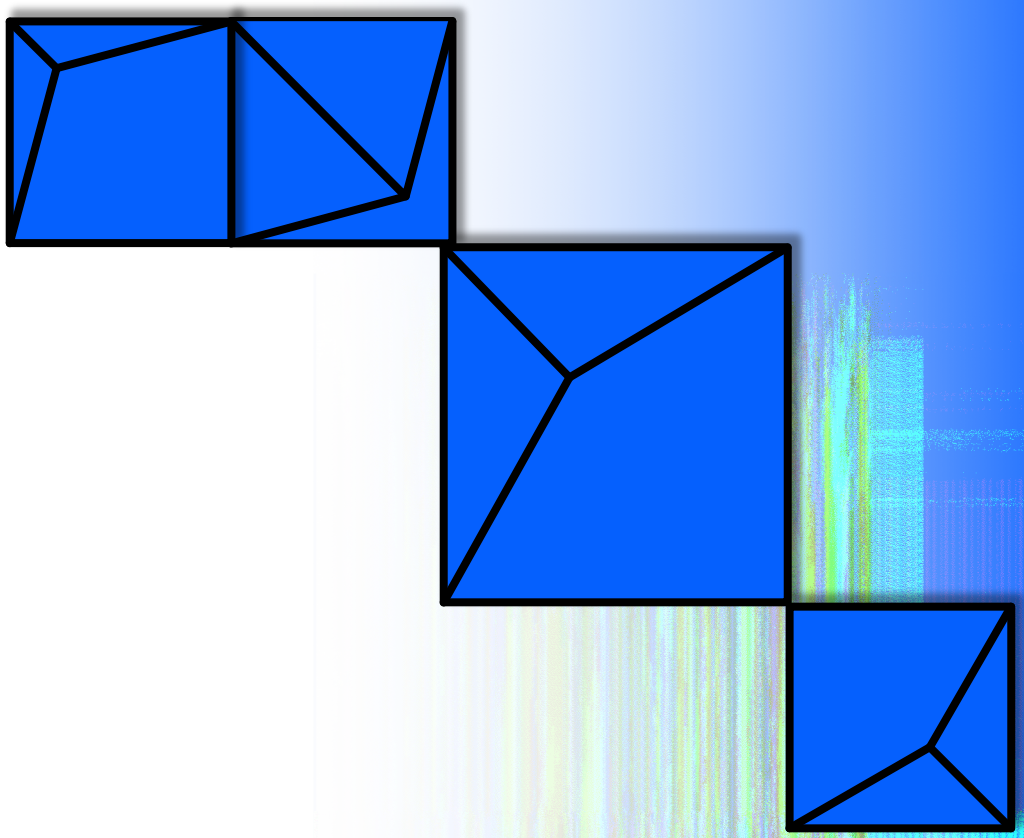
- Remove a REST service from a HTTP server (ZHTPS REMOVE)

Conclusion

- PJ45579 (June 2019):
  - Provides support for path parameters, non-compatible JSON names, XML attributes, and multiple response objects
- PJ45968 (Jan 2020):
  - Makes it easier to manage multiple versions of REST APIs
- PJ45897 (June 2020):
  - Easily create REST artifacts for an OpenAPI document
- PJ46043, PJ46010 (May, Aug 2020):
  - Easily deploy REST services to a test system

Thank You

Questions? Comments?



Trademarks

Notes

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

This presentation and the claims outlined in it were reviewed for compliance with US law. Adaptations of these claims for use in other geographies must be reviewed by the local country counsel for compliance with local laws.