REST versioning:
Methods for handling API changes on z/TPF

2022 TPF Users Group Conference
March 27-30, Dallas, TX
Web Services

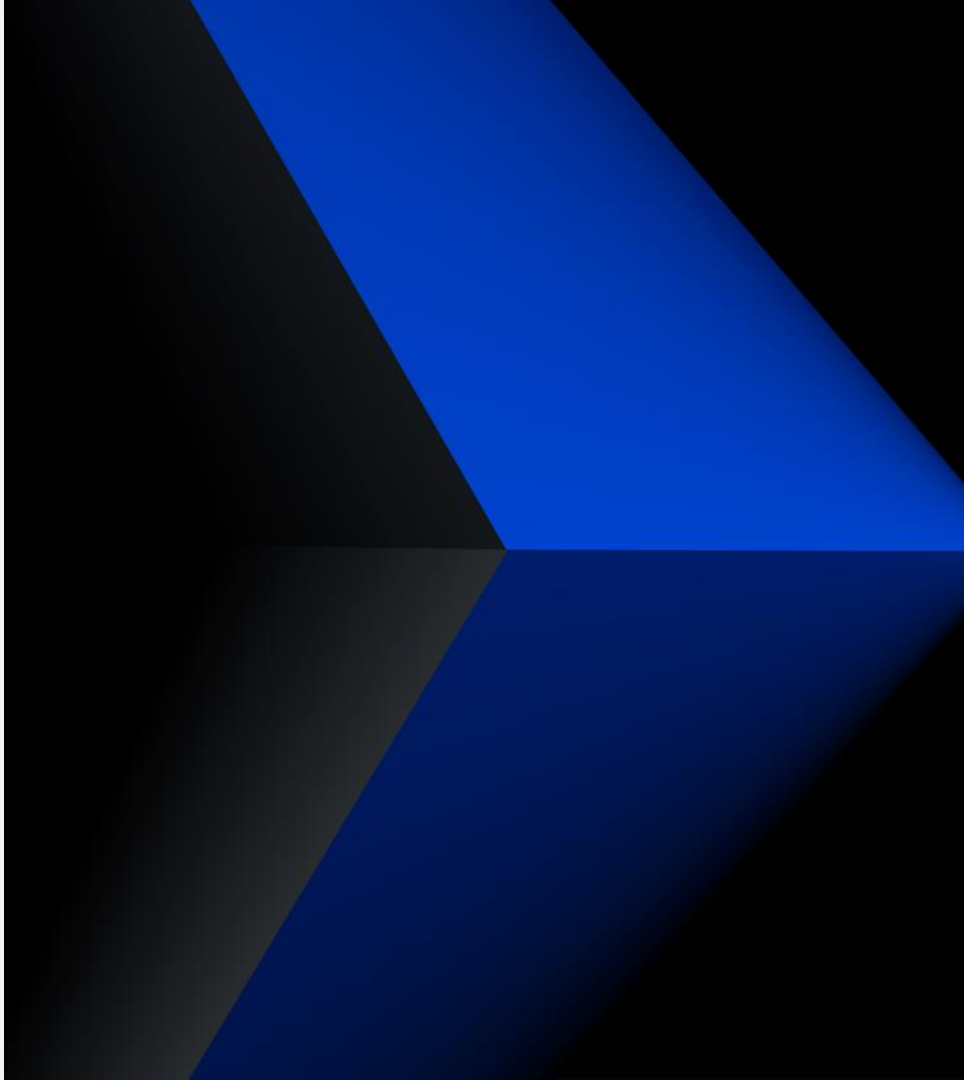
Bradd Kadlecik

IBM Z

IBM

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.



Changing a REST API

Minor update

Backward compatible:
- REST clients are not required to change
- REST provider can maintain 1 version for handling both old and new requests
- Migration handled by server

Optionally provide url for prior versions
/service/ccv
/service/v1r0/ccv
/service/v2r1/ccv

Major update

Interface breaking:
- REST clients must be changed
- REST provider must maintain 2 separate versions to process old requests v new requests
- Migration handled by client

Provide new url for current version
/service/v2/ccv
/service/v1/ccv
/service/v2r0/ccv

REST API change scenarios

Minor update: backward compatible change

Adding a non-required element/property
- at the end
- within an object

Major update: interface breaking change

Removing an element/property
Adding a required element/property
Changing an element/property name
Changing structural layout

Sample REST API version 1.0.0

```
POST http://mytpf:81/tpf/tools/ccv/services
{
  "programs": "QFD*",
  "subsystem": "BSS",
  "terminal": "010000"
}
```

```
typedef struct tpfccvStartReqParms{

  //required fields
  char programs[16];
  char subsystem [4];

  //optional fields
  char terminal[16];

} tpfccvStartReqParms;
```

Sample REST API version 1.1.0

POST http://mytpf:81/tpf/tools/ccv/services

```
{
  "programs": "QFD*",
  "subsystem": "BSS",
  "terminal": "010000",
  "userToken": "JKS"
}
```

```
typedef struct tpfccvStartReqParms{

  //required fields
  char programs[16];
  char subsystem [4];

  //optional fields
  char terminal[16];
  char userToken[8];

} tpfccvStartReqParms;
```

REST API - backward compatible change
1.1.0 v 1.0.0

POST http://mytpf:81/tpf/tools/ccv/services

```
{
 "programs": "QFD*",
 "subsystem": "BSS",
 "terminal": "010000",
 "userToken": "JKS"
}
```

```
D8C6C45C 00000000 00000000 00000000
C2E2E200 F0F1F0F0 F0F00000 00000000
00000000 D1D2E200 00000000
```

POST http://mytpf:81/tpf/tools/ccv/services

```
{
 "programs": "QFD*",
 "subsystem": "BSS",
 "terminal": "010000"
}
```

```
D8C6C45C 00000000 00000000 00000000
C2E2E200 F0F1F0F0 F0F00000 00000000
00000000 00000000 00000000
```

REST API
empty string v no value

POST http://mytpf:81/tpf/tools/ccv/services
```
{
 "programs": "QFD*",
 "subsystem": "BSS",
 "terminal": "010000",
 "userToken": ""
}
```

```
D8C6C45C 00000000 00000000 00000000
C2E2E200 F0F1F0F0 F0F00000 00000000
00000000 00000000 00000000
```

POST http://mytpf:81/tpf/tools/ccv/services
```
{
 "programs": "QFD*",
 "subsystem": "BSS",
 "terminal": "010000"
}
```

```
D8C6C45C 00000000 00000000 00000000
C2E2E200 F0F1F0F0 F0F00000 00000000
00000000 00000000 00000000
```

REST API
null value v no value

POST http://mytpf:81/tpf/tools/ccv/services
```
{
 "programs": "QFD*",
 "subsystem": "BSS",
 "terminal": "010000",
 "userToken": null
}
```

```
D8C6C45C 00000000 00000000 00000000
C2E2E200 F0F1F0F0 F0F00000 00000000
00000000 00000000 00000000
```

POST http://mytpf:81/tpf/tools/ccv/services
```
{
 "programs": "QFD*",
 "subsystem": "BSS",
 "terminal": "010000"
}
```

```
D8C6C45C 00000000 00000000 00000000
C2E2E200 F0F1F0F0 F0F00000 00000000
00000000 00000000 00000000
```

DFDL non-required elements

DFDL elements are not required 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",
   }
   ```

DFDL non-required elements fixed length string

The DFDL for fixed length strings is generated as follows:

<xs:element name="userToken" type="xs:string" dfdl:lengthKind="explicit" dfdl:length="8" dfdl:lengthUnits="bytes" nillable="true" dfdl:useNilForDefault="yes" dfdl:nilKind="literalCharacter" dfdl:nilValue="%NUL;" />

How does one distinguish between an empty string in the request or no value sent? Does it matter? Does one want to allow for null and differentiate it from no value sent?

1) Change textStringPadCharacter to "%SP;" which would mean "" is converted to " "
2) Add a default value to the OpenAPI that would indicate it be treated as no value sent
3) Create a calculated field whose value is set using "{fn:exists(./userToken)}"
4) Use a pointer

DFDL non-required elements
char *

The DFDL for a char * is generated as follows:

<xs:element name="userToken" type="xs:string" tddt:indirectKind="pointer" tddt:indirectLength="8" dfdl:lengthKind="delimited" dfdl:lengthUnits="bytes" dfdl:terminator="%NUL;" dfdl:textTrimKind="none" dfdl:alignmentUnits="bytes" default="" />

- The default value created is a null pointer.
- When pointers are used in DFDL, a null pointer address means the value was not set.
- If the value of either “” or null is set, an address is created.

Adding an optional element at the end

Summary of changes:

1. Update OpenAPI for new property
2. Update DFDL for new element
3. Update C header, assembler DSECT for new field
4. Update app for new field – only need to build segments with code changes

Sample REST API version 1.0.0

```
POST http://mytpf:81/tpf/credit/service
{
  "customer": {
    "name": "Will",
    "surname": "Smith"
  },
  "card": {
    "number": "1111000044442222",
    "expiration": "10/2025"
  },
  "amount": "20.00"
}
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } customer;
  struct {
    char number[16];
    char expiration[8];
  } card;
  long amount;
} creditReqParms;
```

Sample REST API version 1.1.0

```
POST http://mytpf:81/tpf/credit/service
{
  "customer": {
    "name": "Will",
    "surname": "Smith",
    "age": 53
  },
  "card": {
    "number": "1111000044442222",
    "expiration": "10/2025"
  },
  "amount": "20.00"
}
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
    char age;
  } customer;
  struct {
    char number[16];
    char expiration[8];
  } card;
  long amount;
} creditReqParms;
```

REST API – data layout migration minor change

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } customer;
  struct {
    char number[16];
    char expiration[8];
  } card;
  long amount;
} creditReqParms;
```

```
E6899393 00000000 00000000 00000000
E29489A3 88000000 00000000 00000000
F1F1F1F1 F0F0F0F0 F4F4F4F4 F2F2F2F2
F1F061F2 F0F2F500 00000000 000007D0
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
    char age;
  } customer;
  struct {
    char number[16];
    char expiration[8];
  } card;
  long amount;
} creditReqParms;
```

```
E6899393 00000000 00000000 00000000
E29489A3 88000000 00000000 00000000
35F1F1F1 F1F0F0F0 F0F4F4F4 F4F2F2F2
F2F1F061 F2F0F2F5 00000000 00000007
D0
```

Adding an optional element in an object

Summary of changes:

1. Update OpenAPI for new property
2. Update DFDL for new element
3. Update C header, assembler DSECT for new field
4. Update app for new field – rebuild all segments that reference the structure/DSECT

Sample REST API - pointers version 1.0.0

```
POST http://mytpf:81/tpf/credit/service
{
  "customer": {
    "name": "Will",
    "surname": "Smith"
  },
  "card": {
    "number": "1111000044442222",
    "expiration": "10/2025"
  },
  "amount": "20.00"
}
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } *customer;
  struct {
    char number[16];
    char expiration[8];
  } *card;
  long amount;
} creditReqParms;
```

Sample REST API - pointers version 1.1.0

```
POST http://mytpf:81/tpf/credit/service
{
  "customer": {
    "name": "Will",
    "surname": "Smith",
    "age": 53
  },
  "card": {
    "number": "1111000044442222",
    "expiration": "10/2025"
  },
  "amount": "20.00"
}
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
    char age;
  } *customer;
  struct {
    char number[16];
    char expiration[8];
  } *card;
  long amount;
} creditReqParms;
```

Sample REST API – data layout migration minor change

```
typedef struct {
 struct {
  char name[16];
  char surname[16];
 } *customer;
 struct {
  char number[16];
  char expiration[8];
 } *card;
 long amount;
} creditReqParms;
```

```
00000000 18F28720 00000000 18F28E00
00000000 000007D0
0000000018F28720:
E6899393 00000000 00000000 00000000
E29489A3 88000000 00000000 00000000
```

```
typedef struct {
 struct {
  char name[16];
  char surname[16];
  char age;
 } *customer;
 struct {
  char number[16];
  char expiration[8];
 } *card;
 long amount;
} creditReqParms;
```

```
00000000 18F28720 00000000 18F28E00
00000000 000007D0
0000000018F28720:
E6899393 00000000 00000000 00000000
E29489A3 88000000 00000000 00000000
35
```

Adding an optional element in an object w/ pointers

Summary of changes:

1. Update OpenAPI for new property
2. Update DFDL for new element
3. Update C header, assembler DSECT for new field
4. Update app for new field – only need to build segments with code changes

Summary of benefits of using pointers in REST

1. Easy handling of variable size data (strings and arrays).
2. Can generate C headers from OpenAPI documents using tpfrestgen.
3. Can be converted to a contiguous area with offsets for storing or copying using tpf_dfdl_createData, tpf_dfdl_readData.
4. Can be used to indicate whether or not data existed in the request.
5. Can segment the data by objects/structures (similar to C++ object mentality) to minimize recompiles needed for updates.

Do we want to add an option to tpfrestgen to create C structures in such a way?

Sample REST API version 1.0.0 -> 2.0.0

POST http://mytpf:81/tpf/credit/service

```
{
 "customer": {
  "name": "Will",
  "surname": "Smith"
 },
 "card": {
  "number": "1111000044442222",
  "expiration": "10/2025"
 },
 "amount": "20.00"
}
```

POST http://mytpf:81/tpf/credit/v2/service

```
{
 "card": {
  “account": "1111000044442222",
  "expiration": "10/2025"
 },
 "zip": 12601,
 "amount": "20.00"
}
```

Handling major changes different operationIds

Create 2 different swagger documents and use 2 different operationIds:

1.0.0:

credit.swagger.json	-> base path of /tpf/credit
credit.srvc.json	-> operationId of tpfcredit

2.0.0:

creditv2.swagger.json	-> base path of /tpf/credit/v2
creditv2.srvc.json	-> operationId of tpfcreditv2

Handling major changes
same operationId, different version

Create 2 different swagger documents and use 2 different code paths:

1.0.0:

credit.swagger.json	-> base path of /tpf/credit
credit.srvc.json	-> operationId of tpfcredit, version 1.0.0

2.0.0:

creditv2.openapi.json	-> base path of /tpf/credit/v2
creditv2.srvc.json	-> operationId of tpfcredit, version 2.0.0

REST API – data layout migration combined data

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } *customer;
  struct {
    char number[16];
    char expiration[8];
  } *card;
  long amount;
} creditReqParms
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } *customer;
  struct {
    char number[16];
    char expiration[8];
  } *card;
  long amount;
  char version[8];
  unsigned int zip;
} creditReqParms;
```

Capturing the version in the request
DFDL customizations

```
<xs:group name="srvcVersion">
    <xs:sequence>
        <xs:element name="version" type="xs:string" dfdl:length="8"
            dfdl:outputValueCalc="{$nv:ISrvcVersion}"/>
    </xs:sequence>
</xs:group>
```

```
<xs:element name="amount" type="xs:decimal" dfdl:binaryDecimalVirtualPoint="2" default="0"
dfdl:length="8" dfdl:lengthKind="explicit" dfdl:lengthUnits="bytes"/>

<xs:sequence dfdl:hiddenGroupRef="srvcVersion"/>

<xs:element name="zip" type="xs:int" default="0" dfdl:length="4" dfdl:lengthKind="explicit"
dfdl:lengthUnits="bytes"/>
```

Handling major changes common structures

Create 2 different swagger documents and account for differences in application:

1.0.0:

credit.swagger.json -> base path of /tpf/credit
credit.srvc.json -> operationId of tpfcredit, version 1.0.0
creditReqParms.gen.dfdl.xsd -> definition of combined v1/v2 request structure
creditRespParms.gen.dfdl.xsd -> definition of combined v1/v2 response structure

2.0.0:

creditv2.openapi.json -> base path of /tpf/credit/v2
creditv2.srvc.json -> operationId of tpfcredit, version 2.0.0

REST API – data layout migration
use generated structures

```
typedef struct {
  struct {
    char *name;
    char *surname;
  } customer;
  struct {
    char *number;
    char *expiration;
  } card;
  long amount;
} creditReqParms
```

```
typedef struct {
  struct {
    char *account;
    char *expiration;
  } card;
  unsigned int zip;
  long amount;
} creditReqParmsv2;
```

REST API – data layout migration
create structure to structure mapping for 1.0.0

```
typedef struct {
  struct {
    char *name;
    char *surname;
  } customer;
  struct {
    char *number;
    char *expiration;
  } card;
  long amount;
} creditReqParms
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } customer;
  struct {
    char number[16];
    char expiration[8];
  } card;
  long amount;
  unsigned int version;
  unsigned int zip;
} creditReqCommonParms;;
```

REST API – data layout migration
create structure to structure mapping for 2.0.0

```
typedef struct {
  struct {
    char *account;
    char *expiration;
  } card;
  unsigned int zip;
  long amount;
} creditReqParmsv2;
```

```
typedef struct {
  struct {
    char name[16];
    char surname[16];
  } customer;
  struct {
    char number[16];
    char expiration[8];
  } card;
  long amount;
  unsigned int version;
  unsigned int zip;
} creditReqCommonParms;;
```

Handling major changes
use generated structures and s2smap

Create 2 different swagger documents and use 2 different code paths:

1.0.0:

credit.swagger.json -> base path of /tpf/credit
credit.srvc.json -> operationId of tpfcredit, version 1.0.0
creditReqParms.gen.dfdl.xsd -> definition of v1 request structure
creditRespParms.gen.dfdl.xsd -> definition of v1 response structure
creditReqCommonParms.gen.dfdl.xsd -> definition of internal request structure
creditRespCommonParms.gen.dfdl.xsd -> definition of internal response structure

2.0.0:

creditv2.openapi.json -> base path of /tpf/credit/v2
creditv2.srvc.json -> operationId of tpfcredit, version 2.0.0
creditReqParmsv2.gen.dfdl.xsd -> definition of v2 request structure
creditRespParmsv2.gen.dfdl.xsd -> definition of v2 response structure

Summary of making major REST API changes

1. Different OpenAPI documents with different URL base paths can maintain what gets processed for the request and response.

2. Either the same or different operationIds can be used for different versions. The version can be retrieved using name-value pair functions or stored in the request structure. The same can be done with the operationId.

3. Can maintain a common structure that either request or response versions can use. This may result in duplicate places to look for the same data (depending on the version being processed) or having to populate both places where the data can reside.

4. Can use generated artifacts and use a common request/response structure to map to/from.

REST APAR history

1. PJ45968 (Jan 2020) – REST multiversion support (supports .openapi.json file extension)
2. PJ45953 (May 2020) – DFDL support for C constructs (involving pointers)
3. PJ46043 (May 2020) – tpfdfdlgen pointer support
4. PJ45897 (June 2020) – tpfrestgen (zTPF OpenAPI code generation utility)
5. PJ46213 (April 2021) - DFDL structure to structure mapping (interchange pointers and offsets)

Summary of recommendations

1. Update the major,minor version in the OpenAPI document when making changes. This is used to set the ISrvcVersion name-value pair on z/TPF.

2. Use pointers if the application needs to know if a non-required field was passed in the request.

3. Consider using pointers for structure definitions to simplify recompiles needed when adding additional fields in the future.

4. Use the .openapi.json extension for OpenAPI descriptors when you need to maintain more than 1 version of a REST API. This allows for the operationIds (ISrvcName name-value pair) to remain the same for metrics collection.

Thank you