Overview

The OPSDB data store, as used by the Axway API Gateway, is used for efficient writing of events originating within the gateway. This document describes the logical structure of the datastore, the internal, low-level file format used to represent it, and the more stable external export format.

Logical structure of the datastore.

The datastore consists of a sequence of partially ordered events. Events have the following properties:

Property	**Type**	**Meaning**
CorrelationID	Opaque 128-bit string	created for an transaction. The same correlation ID is used for that transaction and all outbound transactions from the gateway carried out as a result of handling it.
Leg	Integer	Within a correlation, the leg identifies the transaction within that correlation that the event is relevant to. The incoming transaction is associated with leg 0 in the correlation. The first outgoing transaction is leg 1, and so on. Some events are not necessarily related to a particular leg in the correlation, so this field is optional.
Tag	String	Each event specifies a tag which identifies the meaning and format of the data. Different parts of the gateway use differen tags to store data relevant to their functionality. The tag indicates the type of event represented by the payload, and by inference, the format of that payload.
Payload	Data	An arbitrary-sized chunk of data. The tag infers its format.

Notes

- All events sharing a single correlation ID are presented in a strictly newest-first order in the datastore, but there is no such guarantee for events with unrelated correlation IDs.
- The correlation ID is generally represented as a 16-byte hex string: however it does have an internal structure of it will be described in the internals section.

Tags currently used include:

Tag	Payload content
“sent”	raw data sent to a remote
“received”	raw data received from a remote
“circuitpath”	JSON data describing the path through a policy
“opevent”	JSON data describing properties of a completed transaction, we’ll refer to these events as “opevent”s
“http”	Internal details of HTTP transaction (binary)

The “opevent” events are associated with a dynamically generated schema that is used to allow a relatively compact representation of the event data without prohibiting extensibility. The schema is stored in a very slowly changing file along side the data files containing the events, and provides a hierarchy of opevent types. For example, there is a “transaction” type, that is used for TCP-based transactions that includes fields such as “bytesSent”, “remoteAddr”. The “http” opevent type extends this “transactions” type, and adds “uri”, “status”, “method” fields.

Physical structure

The datastore is contained in the files located in the groups/group-N/conf/opsdb.d directory. Within this directory, there are a number of .idx files which are used to contain events. There is also a schema.json that contains meta-information for the datastore. The schema can be added to by new modules added to the gateway, but once information is added to the schema it stays there indefinitely. Here’s what the schema.json file looks like.

```
pingu:instance-1/conf/opsdb.d$ jdent < schema.json
{
   "tags": [
      "received",
      "trace",
      "sent",
      "circuitpath",
      "http",
      "opevent",
      "END",
      "sheaders",
      "rheaders"
   ],
   "types": {
```

```
"http": {[1]
   "name": "http",
   "super": "transactions",
   "fields": [
      {
         "name": "uri",
         "type": "TEXT"
      },
      {
         "name": "status",
         "type": "INTEGER"
      },
      {
         "name": "statustext",
         "type": "TEXT"
      },
      {
         "name": "method",
         "type": "TEXT"
      },
      {
         "name": "vhost",
         "type": "TEXT"
      }
   ]
},
"opevent": {
   "name": "opevent",
   "fields": [
      {
         "name": "leg",
         "type": "INTEGER"
      },
      {
         "name": "timestamp",
         "type": "INTEGER"
      },
      {
         "name": "duration",
         "type": "INTEGER"
      },
      {
         "name": "correlationId",
         "type": "TEXT"
      },
      {
         "name": "serviceName",
         "type": "TEXT"
      },
      {
         "name": "subject",
         "type": "TEXT"
      },
      {
```

[1] “jdent” is a small program to intend JSON - you can get it from https://github.com/peadar/JSONdent You can also use “python -m json.tool” as an alternative

```
                "name": "operation",
                "type": "TEXT"
            },
            {
                "name": "type",
                "type": "TEXT"
            },
            {
                "name": "finalStatus",
                "type": "TEXT"
            }
        ]
    },
    "transactions": {
        "name": "transactions",
        "super": "opevent",
        "fields": [
            {
                "name": "bytesSent",
                "type": "INTEGER"
            },
            {
                "name": "bytesReceived",
                "type": "INTEGER"
            },
            {
                "name": "remoteName",
                "type": "TEXT"
            },
            {
                "name": "remoteAddr",
                "type": "TEXT"
            },
            {
                "name": "localAddr",
                "type": "TEXT"
            },
            {
                "name": "remotePort",
                "type": "TEXT"
            },
            {
                "name": "localPort",
                "type": "TEXT"
            },
            {
                "name": "sslsubject",
                "type": "TEXT"
            }
        ]
    }
  }
}
pingu:instance-1/conf/opsdb.d$
```

The schema contains two main parts - first is an array of the names of the tags used in the datastore so far, and the second is the schema for the opevent objects stored therein. When an opevent is added to the datastore, if the type of the opevent is not yet present in the datastore, it is added to the schema, and the schema is rewritten. Note that the number of types of events is very small, so this happens very rarely.

Each opevent type includes a list of typed fields in the opevent, and a supertype. When storing the actual event, it is stored as a JSON array, with the first item in the array being the type of the event, and the second item in the array being a nested array containing the data for each field, eg:

```
[
    "http",
    [
        0, opevent:leg
        1400568590390, opevent:timestamp
        2, opevent:duration
        "0efb7a53c901000024000000c15f0bff", opevent:correlationId
        null, opevent:serviceName
        null, opevent:subject
        null, opevent:operation
        "http", opevent:type
        "Pass", opevent:finalStatus
        251, transactions:bytesSent
        248, transactions:bytesReceived
        "127.0.0.1", transactions:remoteName
        "127.0.0.1", transactions:remoteAddr
        "127.0.0.1", transactions:localAddr
        "49147", transactions:remotePort
        "8080", transactions:localPort
        null, transactions:sslSubject
        "/local/api/test", http:uri
        200, http:status
        "OK", http:statusText
        "DELETE", http:method
        null http:vhost
    ]
]
```

The values are ordered with the fields first by the least derived type of the event, and then as ordered in the schema for that type : the annotations above show the field name assignments for this event, given the schema above.

The .idx files (“index” files) constitute the events themselves. Files are created with a name giving them an ever-increasing serial number, and the one with the highest serial number is considered the “current” file, until that file reaches the target file size configured for the data store.

Each index file is broken into pages. A page is (currently) 1024 * 512 = 524288 bytes in size. The first 512k page of the file is reserved for header information. Pages beyond this are used to store events. Writes to the .idx files are in units of this page size, and the content of each event is stored within a single page. (this places a physical maximum size on the data in an event at a little under 512k)

When a correlation is created, it is assigned to the “current” index file. All events saved against this correlation will be written to this file (this is how we guarantee the ordering of the data within the file - the assignment of space in this file is serialized for all writers)

Physical format of the header page

The header in the first few bytes of the index file is a raw C structure, padded naturally as per the platform’s ABI. Its definition is as follows:

```
struct IndexHeader {
    int magic; // INDEX_MAGIC
    int version; // INDEX_VERSION
    int recordCount;
    int totalCorrelations;
    int activeCorrelations;
    bool clean;
    int64_t unused[64]; // for extensions.
};

enum Constants {
    INDEX_MAGIC = 0xfee1600d,
    INDEX_VERSION = 1
};
```

All other pages (offset in file at pageno * 512 * 1024) have the following header:

```
const uint32_t pageMagic = 0xed6eed6e;
struct Page {
    uint32_t magic;
    uint32_t recordCount;
};
```

The remaining content of the page is made up of a set of event header objects that appear just after that page header, and, at the end of the file, the data payload associated with each event in turn:

```
struct RecordHeader {
    uint32_t offset; // the offset in this page of the record's data.
    size_t tag; // index in schema’s tags array for tag’s name.
    recid prev; // the index in the records array.
    DiskCorrelationID correlationID;
    int16_t leg;
    int16_t flags;
    size_t len;
    enum Flags {
        notstart = 1 << 0,
        notend = 1 << 1
    };
};
```

Each event header contains an “offset” which indicates where the data for this event starts within the page. The first byte beyond the data for record 0 is at offset 524288. The first byte beyond the data for record 1 is at the first byte of data in record 0. This means that as the event headers extend towards higher offsets in the page, the data payload extends to lower offsets, until eventually there is not enough space to store a new event in the page. At that point, the page is considered full, and the next event is written to a new page in the file.

To complete the current binary interface, the following definitions are required:

```
struct recid {
    size_t page;
    size_t record;
}

struct DiskCorrelationID {
    uint32_t time; // seconds
    uint32_t seq;
    uint32_t opref;
    unsigned char rand[4];
}
```

The recid is used to link events within a single correlation - with each event, we store the page ID, and record offset within that page of the data.

The correlation ID above is what we use to derive the correlation ID hex string - to do so, we treat the structure as an array of unsigned bytes, and generate a 2-byte hex sequence for each byte. the structure includes the current timestmap (seconds since Jan 1, 1970), a sequence within that second, an “opref”, and 4 random bytes of data.

The “opref” is used as an optimization currently - it contains the number of the index file containing the correlation. So, given a correlation ID, we can immediately pinpoint the file which contains that correlation ID, and scan it for events associated with that ID.

Extracting information from the files, part 1: VTD

The VTD executable works on the raw data files to query the content. To use vtd, first add the <install directory>/posix/bin directory to your path:

```
$ pwd
/home/peadar/Axway-7.3.1/apigateway
$ export PATH=$PWD/posix/bin:$PATH
$
```

Then navigate to the opsdb.d directory:

```
$ cd groups/group-2/instance-1/conf/opsdb.d
$
```

The easiest command to run is “list”: this lists the correlation IDs of all correlations in the data store:

```
$ vrun vtd list | head -10
15fb7a5302020000024000000802e5d47
14fb7a530002000024000000aeee601c
14fb7a53fe0100002400000096895bdb
14fb7a53fc0100002400000089b19ec3
13fb7a53f901000024000000d1625ce7
13fb7a53f801000024000000e0f213c9
13fb7a53f5010000240000000baaa291
13fb7a53f4010000240000009831e517
12fb7a53f10100002400000004d5ffc4
12fb7a53f00100002400000031320451
```

Given a particular correlation ID, you can dump out the details of everything known about that correlation ID. Note that because the details are somewhat dependent on the schema, that is also dumped as part of the output.

```
$ vrun vtd info 14fb7a53fe0100002400000096895bdb | jdent
{
  "schema": { … }
  "correlation": [
        {
```

```
            "correlationId": "14fb7a53fe0100002400000096895bdb",
            "leg": 0,
            "tag": "received",
            "offset": 179903,
            "len": 247,
            "prev": {
                "page": 0,
                "record": 0
            },
            "flags": 0,
            "data": "OPTIONS /step9/test HTTP/1.1\r\nAccess-Control-Request-Headers:
X-Hello-World, X-Request-Value\r\nAccess-Control-Request-Method: POST\r\nOrigin:
step9.api.qa.vordel.com\r\nUser-Agent: Jakarta Commons-HttpClient/3.1\r\nHost:
marriott.apis.vordel.com:9088\r\n\r\n"
        },
        {
            "correlationId": "14fb7a53fe0100002400000096895bdb",
            "leg": 0,
            "tag": "sent",
            "offset": 179370,
            "len": 533,
            "prev": {
                "page": 13,
                "record": 1699
            },
            "flags": 0,
            "data": "HTTP/1.1 200 OK\r\nDate: Tue, 20 May 2014 06:49:56 GMT\r\nAllow:
DELETE, ...        },
        {
            "correlationId": "14fb7a53fe0100002400000096895bdb",
            "leg": -1,
            "tag": "circuitpath",
            "offset": 179126,
            "len": 244,
            "prev": {
                "page": 13,
                "record": 1700
            },
            "flags": 0,
            "data": "[ { \"policy\": \"API Broker\", \"execTime\": 0, \"filters\": [  {
\"name\": \"Options Request\", \"type\": \"ApiShuntFilter\", \"class\":
\"com.vordel.apiportal.runtime.broker.ApiShuntFilter\", \"status\": \"Pass\",
\"filterTime\": 1400568596553, \"execTime\": 0 }  ] } ]"
        },
        {
            "correlationId": "14fb7a53fe0100002400000096895bdb",
            "leg": 0,
            "tag": "opevent",
            "offset": 178805,
            "len": 209,
```

```
            "prev": {
                "page": 13,
                "record": 1702
            },
            "flags": 0,
            "event": [
                "http",
                [
                    0,
                    1400568596552,
                    1,
                    "14fb7a53fe0100002400000096895bdb",
                    null,
                    null,
                    null,
                    "http",
                    "Pass",
                    533,
                    247,
                    "10.142.59.169",
                    "10.142.59.169",
                    "10.142.56.36",
                    "4166",
                    "9088",
                    null,
                    "/step9/test",
                    200,
                    "OK",
                    "OPTIONS",
                    "Step9"
                ]
            ]
        },
    ]
```

Here we get a list of correlations, each with a tag and a leg. Depending on the tag, we present either an “event” or “data” payload within the correlation. We can also access events related to a specific tag, eg, to see all the data “sent” in a correlation for leg 0, or the circuit path for the correlation:

```
$ vrun vtd stream 14fb7a53fe0100002400000096895bdb sent 0
HTTP/1.1 200 OK
Date: Tue, 20 May 2014 06:49:56 GMT
Allow: DELETE, GET, HEAD, OPTIONS, POST, PUT
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: x-hello-world, x-request-value
Access-Control-Allow-Methods: OPTIONS, POST
Access-Control-Allow-Origin: step9.api.qa.vordel.com
Access-Control-Max-Age: 86400
```

```
Server: Gateway
Connection: close
X-CorrelationID: Id-14fb7a53fe0100002400000096895bdb 0
Host: marriott.apis.vordel.com:9088
User-Agent: Jakarta Commons-HttpClient/3.1
Content-Type: text/plain

$ vrun vtd stream 14fb7a53fe0100002400000096895bdb circuitpath | jdent
[
    {
        "policy": "API Broker",
        "execTime": 0,
        "filters": [
            {
                "name": "Options Request",
                "type": "ApiShuntFilter",
                "class": "com.vordel.apiportal.runtime.broker.ApiShuntFilter",
                "status": "Pass",
                "filterTime": 1400568596553,
                "execTime": 0
            }
        ]
    }
]
$
```

You can use the “events” command to show a more friendly formatted version of the opevent entries (this will use the schema to properly name the fields in the opevent objects)

```
$ vrun vtd events 14fb7a53fe0100002400000096895bdb | jdent
[
    {
        "uri": "/step9/test",
        "status": 200,
        "statustext": "OK",
        "method": "OPTIONS",
        "vhost": "Step9",
        "bytesSent": 533,
        "bytesReceived": 247,
        "remoteName": "10.142.59.169",
        "remoteAddr": "10.142.59.169",
        "localAddr": "10.142.56.36",
        "remotePort": "4166",
        "localPort": "9088",
        "sslsubject": null,
        "leg": 0,
        "timestamp": 1400568596552,
        "duration": 1,
        "correlationId": "14fb7a53fe0100002400000096895bdb",
        "serviceName": null,
```

```
        "subject": null,
        "operation": null,
        "type": "http",
        "finalStatus": "Pass"
    }
]
```

Finally, you can dump the entire database in JSON format to a text file, eg:

```
pingu:instance-1/conf/opsdb.d$ vrun vtd  dump  | jdent
[
    {
        "correlationId": "15fb7a530202000024000000802e5d47",
        "leg": -1,
        "tag": "END",
        "offset": 176113,
        "len": 0,
        "prev": {
            "page": 13,
            "record": 1715
        },
        "flags": 0,
        "data": ""
    },
    {
...
```

This will dump the entire data file into JSON formatted text. This is intended for use during upgrade, and provides a more stable format for interpreting the content than the raw binary files.

Extracting information from the files, part 2: REST API

The preferred way to access the opsdb data is via the REST interface as used by the traffic monitor web UI. Accessing a gateway's traffic monitor API requires that you go through the node manager router interface. The gateway itself exposes the interface under "/ops" on its management port. However, access to this management port is restricted to the local node manager, so you must use the following:

https://localhost:8090/api/router/service/instance-1/ops/

The REST API is documented with the API Gateway - an up-to-date version of the documentation is available on demand

Extracting information from the files, part 3: Creating an offline REST API

The REST interface is provided by the vshell service, which is used to host the gateway. However, a much smaller process can be used to serve _just_ the REST interface, once it has access to the physical data for the datastore (located in the gateway instance's conf/opsdb.d directory.)

First, you need to configure the service. Create a file, opsdb.xml, with the following content:

```
<NetService provider="NetService">

    Basic configuration to provide opsdb "traffic monitor" database REST
    API on a directory.

    First, we say where we want trace information to go:
    <FileTrace filename="@stdout"/>

    If vshell is started with "-Ddebug", then enable DEBUG level trace.
    <if property="debug">
        <SystemSettings tracelevel="DEBUG"/>
    </if>

    Service HTTP requests
    <HTTP provider="HTTP" name="Foo">
        Listen on port 8880: <InetInterface port="8880"/>

        handle the traffic monitor API at "/ops", using the files in
        /tmp/opsdb-snap for the data:
        <OPDbViewer provider="HTTPops" uriprefix="/ops/" appendDb="/tmp/opsdb-snap"/>

    </HTTP>

</NetService>
```

Once saved, you can start vshell with this file:

```
$ $GWDIR/posix/bin/vrun vshell ./opsdb.xml
INFO    09/Jul/2014:17:17:42.006 [928e2740] TCP interface
INFO    09/Jul/2014:17:17:42.006 [928e2740]     checking invariants for interface *:8880
INFO    09/Jul/2014:17:17:42.006 [928e2740]         listen on address: 0.0.0.0:8880
INFO    09/Jul/2014:17:17:42.008 [928e2740] operations DB responder waiting for requests
on /ops/
INFO    09/Jul/2014:17:17:42.008 [928e2740] starting 4 idle netsvc threadpool threads.
Max 1024
INFO    09/Jul/2014:17:17:42.008 [928e2740] service started (version 7.3.1-PeterSource,
pid 21609)
```

With the vshell service running, you can now access the OPS API directly:

```
$ curl 'http://localhost:8880/ops/search?format=json&field=status&value=200&op=ne'
{"processId":"", "data":[
{"uri":"/test","status":403,"statustext":"Forbidden","method":"GET","vhost":null,"bytesSe
```

nt":477,"bytesReceived":366,"remoteName":"127.0.0.1","remoteAddr":"127.0.0.1","localAddr"
:"127.0.0.1","remotePort":"57759","localPort":"8080","sslsubject":null,"leg":0,"timestamp
":1404919843693,"duration":1,"correlationId":"2360bd536e00000000000000537a7362","serviceN
ame":null,"subject":null,"operation":null,"type":"http","finalStatus":"Fail"
        },

{"uri":"/test","status":403,"statustext":"Forbidden","method":"GET","vhost":null,"bytesSe
nt":451,"bytesReceived":340,"remoteName":"127.0.0.1","remoteAddr":"127.0.0.1","localAddr"
:"127.0.0.1","remotePort":"57035","localPort":"8080","sslsubject":null,"leg":0,"timestamp
":1404918342955,"duration":2,"correlationId":"465abd5313000000000000002cd99d36","serviceN
ame":null,"subject":null,"operation":null,"type":"http","finalStatus":"Fail"
        },

{"uri":"/hello","status":403,"statustext":"Forbidden","method":"GET","vhost":null,"bytesS
ent":451,"bytesReceived":341,"remoteName":"127.0.0.1","remoteAddr":"127.0.0.1","localAddr
":"127.0.0.1","remotePort":"56941","localPort":"8080","sslsubject":null,"leg":0,"timestam
p":1404917585785,"duration":30,"correlationId":"5157bd53050000000000000019a89821","servic
eName":null,"subject":null,"operation":null,"type":"http","finalStatus":"Fail"
        }
]}%