Server Functional Spec

Release 0.7

Table of Contents

1Schemata Management 2

2Extension Management 3

3API Reference 4

3.1 General 4

3.2 Methods 4

3.2.1 /event 4

3.3 /session 5

3.3.1 GET /session/:id 5

3.3.2 PUT /session/:id 7

4Future 8

# Connections

Connections represent a logically persistent, non-expiring communication channel between a client address space and an experiment schema on the server. All communication between a client and the server, is carried out over a connection, which must be opened first.

A client process should only need one connection because it’s hard to imagine a case when a component will want to participate in experiments from more than one schema.

Server is configured with a fixed number of active connections:

variant.max.connections = 100 // default

After this many active connections, a request for new connection will receive a too many connections error. Connections do not expire, so clients must take measures to avoid connection leak, e.g. ensure that a connection is closed when a client process is terminated. This will ensure that Variant server can operate without a restart even if client applications crash.

TODO: Client should close an idle connection after a configurable period of inactivitiy.

A connection is bound to a schema. All connections to the same schema (called parallel connections) are physically separate and each takes up a slot in the connection table. But they are related in the sense that sessions are shared between them: a session opened in one, is accessible in any parallel connection. This is necessary for sharing session state between different instances of the client, e.g. different processes.

When a connection is closed by the client:

* Client expires all cached sessions associated with this connection, marks connection as closed by client and unlinks it from the connection list. If client code holds on to the connection object and attempts to operate on it or on a derived object, client will throw an exception without going to the server.
* Server to expire all server sessions, associated with this and let the vacuum thread to clean them out. (This is a bug. See the TODO above)
* Server removes the connection object from the connection table.

When a connection is closed by the server due to schema redeployment:

* The connections are drained: existing sessions continue to operate until organically expired against an in-memory read-only snapshot of the schema.
* All new connections to this schema go against the new version of the schema.

# Sessions

A session is created in a particular connection and this association

# Schemata Management

## General

Schema files are read once during server startup from a file located inside the schema deploy directory. If a fatal error is encountered, the server must be restarted after it is corrected. Until then, all subsequent connection request will receive HTTP status 503 “Service Unavailable.”

Only a single schema file per server, e.g. per deploy directory, is supported at this time.

The schema file must start with the meta section:

{

"meta": {

"name": schema-name::NameString,

"comment": schema-comment::String?

"hooks": [*hook-def:Object*,...]

},

“states”: {...},

“tests”: {...)

}

## Schema deploy directory location

Schema files are located in the OS directory specified by, in the order of significance

* -Dvariant.schemas.dir system property
* variant.data.dir configuration property
* (/schemas classpath directory — future improvement?)

Value is treated the same as Java’s File(String), i.e. if starts with slash is understood as absolute path, otherwise as relative to the applicatioin’s running directory.

# Server Extension API

## Pluggable Event Flushers

As of 0.7.0:

Event flusher class is configured by the variant.event.flusher.class.name and variant.event.flusher.class.init config keys. The class must be on the runtime classpath, e.g. in a jar file inside the /lib directory.

## User Hooks

User hooks are handlers (or listeners) for variant lifecycle events (LSEs). They can be schema, session, state or test scoped.

|  |  |  |  |
| --- | --- | --- | --- |
| Lifecycle Event | Schema Scope | Runtime Scope | When fires |
| State Parsed | Global | Schema | At schema parse time, when a state has been successfully parsed. |
| Test Parsed | Global | Schema | At schema parse time, when a test has been successfully parsed. |
| Schema Parsed | Global | Schema | At schema parse time when a new schema has been successfully parsed. |
| Schema Deployed (future) | Global | Schema | Right after new schema has been deployed. |
| Schema Undeployed (future) | Global | Schema | Right before an active schema is undeployed. |
| Test Qualification | Test | State Request | When a session is about to be qualified for a test. |
| Test Targeting | Test | State Request | When a session is about to be targeted for a test. |
| Session Created (future) | Global | Session | When a session is about to be targeted for a test. |
| Session Destroyed (future) | Globoal | Session | When a session is about to be targeted for a test. |

### Definition

User hooks are registered in the schema. Hooks, whose domain is Schema, must be defined in the meta section. Hooks with the Test domain must be defined at the test level.

"meta": {

...

"hooks": [*hook-def:Object*,...]

}

"tests": [{

...

"hooks": [*hook-def:Object*,...]

}]

hook-def := {

“name”: hook-name::NameString,

“class”: fully-qualified-class-name::String

“init”: init-parameters::JSON-String ?

}

It is a parse time error to define schema-domained hooks at a test level or test-domained hooks at the meta level. Names are required and must be unique within the domain.

The value of the ‘class’ property must be the fully qualified class name, as returned by the Class.getName() method.

The value of the ‘init’ property is an arbitrary JSON literal. It will be passed to the UserHook.init() method immediately after instantiation, as a com.typesafe.comfig.ConfigValue instance, which represents any JavaScript literal.

### Execution

If more than one hook is registered for an LSE, they form a listener chain and are posted in the ordinal order, i.e. order they are defined. For schema and session scoped hooks, this means the order in which they are mentioned in the “hooks” clause. For the state and test scoped hooks this means the order in which they are mentions in the hook-refs clause of the particular state or test.

For each hook in a chain, if the post() method returns a non-null, the remaining listeners are ignored. Otherwise, the next listener on the chain is posted. If no user hook returned a value, the default hook is posted. Each user LSE type provides a default hook, which is posted if either no user hooks were registered or none returned a value. By contract, default hook will not return null.

# API Reference

## General

* Property names are case insensitive, i.e. createDate is the same as CreateDate.
* Sessions are stored on the server as serialized JSON strings and are deserialized lazily, if server needs them.

## Methods

Notation:

"name": <Number?=NOW>

* **"name":** name to the left of colon is the literal property name. (
* **<Number?=NOW>** information inside the angle brackets is the data type (String/Number/Boolean/Array/Object), optionally followed by the question mark, indicating that this field is optional, optionally followed by the equal sign and the default value. If no question mark, this field is required. If question mark, but no equal sign, the field is optional with no default.
* Note, that the type specification

### /connection

#### POST /connection/:schema-name

Open a new connection to an XDM schema.

|  |  |  |
| --- | --- | --- |
| Request Headers | | |
| Content-Type | text/plain; charset=utf-8 | |
| Request Body | | |
| None. | | |
| Response Headers | | |
|  | | |
| Response body | | |
| Connection.toJson() | | |
| Response Codes | | |
| 200 OK | | Connection created. |
| 400 BAD\_REQUEST | | Any error. Standard error JSON in response body. |

#### DELETE /connect/:schema-id

Close an existing connection to an XDM schema.

|  |  |  |
| --- | --- | --- |
| Request Headers | | |
| Content-Type | text/plain; charset=utf-8 | |
| Request Body | | |
| None. | | |
| Response Headers | | |
|  | | |
| Response body | | |
| None | | |
| Response Codes | | |
| 200 OK | | Connection created. |
| 400 BAD\_REQUEST | | Any error. Standard error JSON in response body. |
|  | |  |

### /event

#### POST /event

Trigger a user-defined event in the specified session.

|  |  |  |  |
| --- | --- | --- | --- |
| Request Headers | | | |
| Content-Type | | text/plain; charset=utf-8 | |
| Request Body | | | |
| {  "sid": <String>,  "name": <String>,  "value": <String?>,  "ts": <Number?=NOW>,  "params": <Array?> [  {  "key": <String>,  "val": <String?>  },  ...  ],  } | | | |
| sid | Current variant session ID. | | |
| name | The name of the event. Any string. | | |
| value | Value of the event. Any string. | | |
| ts | Event creation date, as Epoch time. Optional. If not given, defaults to now. | | |
| params | A map of event parameters. Optional. | | |
| Response Headers | | | |
|  | | | |
| Response body | | | |
| None. | | | |
| Response Codes | | | |
| 200 OK | | | Event triggered. |
| 400 BAD\_REQUEST | | |  |

### /session

|  |  |
| --- | --- |
| GET /session/:id | |
| Get existing session by ID.   |  |  |  | | --- | --- | --- | | Headers | | | | Content-Type | text/plain; charset=utf-8 | | | Request Body | | | | None | | | | Response body | | | | CoreSession.toJson() | | | | Response Headers | | | |  | | | | Response Codes | | | | 200 OK | | Session found and returned. | | 400 BAD\_REQUEST | | Any error, e.g. session expired. Standard error JSON in response body. |  PUT /session | |
| Save (create or replace) user session by session ID. Idempotent. If session did not exist, it will be created in the given connection (must exist). Otherwise, given connection must match that in which the original session was created. |  |

|  |  |  |
| --- | --- | --- |
| Headers | | |
| Content-Type | text/plain; charset=utf-8 | |
| Request Body | | |
| {  [cid:<connection-id](cid:%3cconnection-id)>,  ssn:CoreSession.toJson()  } | | |
| Response body | | |
| None. | | |
| Response Codes | | |
| 200 OK | | Session created or replaced. |
| 400 BAD\_REQUEST | | Any error, e.g. session expired. Standard error JSON in response body. the HTTP Status header. |

# Future

## Schema Management

* Symbolic variables in schema definition file.
* Multi-line comments.
* Write own parser to provide support for:
  + Line numbers for semantical errors.
  + Preserve original line number before removal of comments.
* Expire connections that appear to have leaked, e.g. have no active sessions for some period of time.
* Hard vs soft schema reload. Hard will recreate an existing schema even if active connections exist, while soft will wait for all active connections to close (possibly indefinitely, i.e. recommended in conjunction with previous point). The goal is to have a mode in which no active sessions will receive a ConnectionClosed exception.