Overview

General Information

Unified API version 22.1.0 works with CPS 18.0+.

Release Notes

CSCvf86865 - Added new parameter called `retrieveAll` in QuerySessionRequest (default: true) to retrieve all or one session entry. Added logic to first check whether the session entry exists for the given key in memcache. If found, then query the particular DB shard.

Default URLs

HA

Table 1: URLs - HA

https://lbvip01:8443/ua/soap	endpoint
https://lbvip01:8443/ua/wsdl/UnifiedApi.wsdl	retrieves the WSDL
https://lbvip01:8443/ua/wsdl/UnifiedApi.xsd	retrieves the XSD (schema)

Audit History

The Audit History is a way to track usage of the various GUIs and APIs it provides to the customer.

If enabled, each request is submitted to the Audit History database for historical and security purposes. The user who made the request, the entire contents of the request and if it is subscriber related (meaning that there is a networkId value), all networkIds are stored as well in a searchable field.

Capped Collection

The Audit History uses a 1 GB capped collection in Mongo Db by default. The capped collection automatically removes documents when the size restriction threshold is hit. The oldest document is removed as each new document is added. For customers who want more than 1 GB of audit data, please contact the assigned Cisco Advanced Services Engineer to get more information.

Configuration in Policy Builder is done in GB increments. It is possible to enter decimals, for example, 9.5 will set the capped collection to 9.5 GB.

Note **PurgeAuditHistoryRequests**

When using a capped collection, Mongo Db places a restriction on the database and does not allow the deletion of data from the collection. Therefore, the entire collection must be dropped and re-created. This means that the PurgeAuditHistory queries have no impact on capped collections.

Note **AuditRequests**

As a consequence of the XSS defense changes to the API standard operation, any XML data sent in an AuditRequest must be properly escaped even if inside CDATA tags. For example, <ExampleRequest>...</ExampleRequest> See AuditType for more information.

Operation

The Audit History can be turned off, but it is on by default.

- **ua.client.submit.audit=true** - property used by Policy Builder and set in `/etc/broadhop/pb/pb.conf`
- **Submit Requests to Audit Log** - Unified API plugin configuration in Policy Builder

Initial Setup

There are three parts to the Audit History

- Server - database and Unified API
- Policy Builder
- Audit Client - bundle that the Policy Builder uses to send Audit requests

To setup the system:

Step 1 Start the Policy Builder with the following property: **-Dua.client.submit.audit=false** (set in `/etc/broadhop/pb/pb.conf`)

Step 2 Add and configure the appropriate plugin configurations for Audit History and Unified API

Step 3 Publish the Policy Builder configuration

Step 4 Start the CPS servers.

Step 5 Restart the Policy Builder with the following property:

-Dua.client.submit.audit=true

-Dua.client.server.url=http://ADDRESS OF SERVER:PORT

Read Requests

The Audit History does not log read requests by default.

- GetRefDataBalance
- GetRefDataServices
- GetSubscriber
- GetSubscriberCount
- QueryAuditHistory
- QueryBalance
- QuerySession
- QueryVoucher
- SearchSubscribers

The Unified API also has a Policy Builder configuration option to log read requests which is set to false by default.

APIs

All APIs are automatically logged into the Audit Logging History database, except for QueryAuditHistory and KeepAlive. All Unified API requests have an added Audit element that should be populated to provide proper audit history.

Querying

The query is very flexible - it uses regex automatically for the id and dataid, and only one of the following are required: id, dataid, or request. The dataid element typically will be the networkId (Credential) value of a subscriber.

Note **Disable Regex**

The use of regular expressions for queries can be turned off in the Policy Builder configuration.

The id element is the person or application who made the API request. For example, if a CSR logs into Control Center and queries a subscriber balance, the id will be that CSR's username.

The dataid element is typically the subscriber's username. For example, if a CSR logs into Control Center and queries a subscriber, the id will be that CSR's username, and the dataid will be the subscriber's credential

(networkId value). For queries, the dataid value is checked for spaces and then tokenized and each word is used as a search parameter. For example, "networkId1 networkId2" is interpreted as two values to check.

The fromDate represents the date in the past from which to start the purge or query. If the date is null, the api starts at the oldest entry in the history.

The toDate represents the date in the past to which the purge or query of data includes. If the date is null, the api includes the most recent entry in the purge or query.

Purging

The Audit History database is capped at 1 GB by default. Mongo provides a mechanism to do this and then the oldest data is purged as new data is added to the repository. There is also a PurgeAuditHistory request which can purge data from the repository. It uses the same search parameters as the QueryAuditHistory and therefore is very flexible in how much or how little data is matched for the purge.

Note **Regex Queries!**

Be very careful when purging records from the Audit History database. If a value is given for dataid, the server uses regex to match on the dataid value and therefore will match many more records than expected. Use the QueryAuditHistory API to test the query.

Control Center

The Control Center version 2.0 automatically logs all requests.

Policy Builder

The Policy Builder automatically logs all save operations (Publish and Save to Client).

HTTP KeepAlive

HA

https://lbvip01:8443/ua/soap/keepalive	Response: <html><body> Keep Alive </body></html>

Dates

Dates are in Zulu/UTC timestamp format. For proper server operation, you must use a consistent timestamp format for all dates in the API requests.

Parsing format: yyyy-MM-ddTHH:mm:ss[.SSS][Z|(+|-)hh:mm]

Note **Hijri Dates**

The Unified API does not support Hijri date translation at this time.

Valid Timestamp Formats

These are the valid timestamps based on the parsing format listed above:

2010-09-30T00:00:00Z	the Z indicates Zulu/UTC time - the database translates to UTC offset of locale/timezone
2010-09-30T00:00:00+00:00	manual inclusion of UTC offset of locale/timezone - the database will store as is
2010-09-30T00:00:00.000Z	uses milliseconds for extra specificity of time
2010-09-30T00:00:00.000+00:00	uses milliseconds for extra specificity of time
2010-09-30T00:00:00	no timezone offset or Z indication - the database translates to UTC offset of locale/timezone unless qns.conf param is set
2010-09-30T00:00:00.000	no timezone offset or Z indication and uses milliseconds for extra specificity of time - the database translates to UTC offset of locale/timezone unless qns.conf param is set

Note **No Timezone Offset or Z - qns.conf parameter**

The Unified API now supports dates that do not include a timezone offset or Z indication for UTC time. When a date is sent that does not include a timezone offset or Z, the API assumes Z/UTC unless the following qns.conf param is set:

-Dua.date.converter.timezone.offset. The timezone offset takes the form of (+|-)hh:mm. For example, -06:00 is Mountain Daylight Time (MDT) while -07:00 is Mountain Standard Time (MST) and +00:00 is UTC:

-Dua.date.converter.timezone.offset=-06:00

Database Timestamp Translation

The database always uses yyyy-MM-ddTHH:mm:ss.SSS(+|-)hh:mm for formatting dates. The database also always translates the datetime to the local server timezone. Therefore, the results may be unexpected if you pass the value with the Z format. For example: 2010-10-15T00:00:00Z produces Thu Oct 14 2010 18:00:00 GMT-06:00 (MDT) in the database instead of Fri Oct 15 2010 00:00:00 GMT-06:00 (MDT) assuming your server is set to North American Mountain Time. The reason is that the server reads the timezone of the incoming value as UTC (+00:00 indicated by the Z) and then translates that to the local server timezone which in this example is North American Mountain Time or -06:00. Mountain Time is 6 hours before (-) UTC.

Daylight Savings Time

Daylight Savings Time adds another wrinkle to the processing of dates and times. In the course of a year, the server timezone will automatically shift from standard time to daylight savings time. In our examples, that would be North American Mountain Daylight Savings Time (MST instead of MDT). During that period, the following will happen:

2010-11-15T00:00:00-06:00 produces Sun Nov 14 2010 23:00:00 GMT-07:00 (MST) in the database

Because Nov 15th 2010 is after the switch back to Standard time in the Mountain Zone but the database is translating to daylight savings time which is an hour earlier. To get Mon Nov 15 2010 00:00:00 GMT-07:00 (MST) during the Daylight Savings portion of the year, you need to pass in 2010-11-15T00:00:00-07:00.

Wikipedia ISO 8601

If you want to understand the ISO 8601 standard for date time handling, the following Wikipedia article could be useful.

http://en.wikipedia.org/wiki/ISO_8601

Services and Service Schedules

Service Schedules use a traditional concept of cron taken from the Quartz package. Quartz documents cron here. While we are not using the CronTrigger class, the explanation of the fields and how the values operate is useful information.

We essentially are trying to model a start and end date and a start and end time using the XML structure shown below. We do not deal with seconds or milliseconds which cron does. Instead we start at minute specificity. Instead of using pure cron notation, we have a startTime and endTime that makes it more human readable. We also use startDate and endDate which along with the start and end times, create the period of time over which the service is active. Then the repeat object handles how the schedule repeats within the specified date/timeframe. The repeat elements use actual Quartz cron notation.

startTime must be before endTime since it represents a range of time within a given day for the service to be active and it is used to build the cron object during processing.

Note **Service Evaluation "Gaps" - Seconds/Milliseconds**

The cron processing appends :59:999 (59 seconds and 999 milliseconds) to the endTime value which means that if you set the endTime to 12:59, the cron processing evaluates that as 12 hours 59 minutes 59 seconds and 999 milliseconds. This helps ensure that service evaluation for start and stop times does not have any "gaps". This is necessary for processing schedules like in the example below which cross date boundaries.

Note **Crossing Date Boundaries**

Schedules are tricky because they operate on two levels:

1. a date period for which the service is active
2. a time period for any given day which the service is active, meaning that you cannot cross date boundaries with the startTime and endTime

The typical gotcha scenario is trying to have a service be active for 24 hours across 2 days - for example, the service starts at 2 am on day 1 and ends at 2 am on day 2.

Most people try to do the following:

Schedule 1: startDate: 2013-10-03, endDate: 2013-10-04, startTime: 02:00, endTime: 01:59

The above does not work because the endTime is before the startTime which will result in an invalid cron object for processing.

Instead, do the following:

Schedule 1: startDate: 2013-10-03, endDate: 2013-10-03, startTime: 02:00, endTime: 23:59

Schedule 2: startDate: 2013-10-04, endDate: 2013-10-04, startTime: 00:00, endTime: 01:59

The above sets 2 schedules for the service: the first is valid from 2:00 am to 11:59 pm on October 3rd and the second is valid from 12:00 am to 1:59 am on October 4th. This creates a 24 hour active period for the service.

Note **Quartz Documentation**

As the Quartz documentation mentions, dayOfMonth and dayOfWeek are related and only one of the fields can contain ? for any given cron expression. dayOfMonth and dayOfWeek both cannot be configured at the same time. Only one parameter can be configured at any given point of time. If configuring dayOfWeek, set ? in dayOfMonth as <dayOfMonth>?</dayOfMonth>. If configuring dayOfMonth, set ? in dayOfWeek as <dayOfWeek>?</dayOfWeek>

While there is a great of flexibility with the current data structure, it is strongly recommended that you fill in all 4 values if you decide to include a repeat element.

field	regex	default
dayOfMonth	[\-,0-9*\?LW/]*	*
month	[\-,0-9*A-Z/]*	*
dayOfWeek	[\-,0-9*\?L#/]*	?
year	[\-,0-9*/]*	*

```
<schedule>
  <startDate>2011-01-01T00:00:00Z</startDate>
  <endDate>2012-01-01T00:00:00Z</endDate>
  <state>ON</state>
  <startTime>00:00</startTime>
```

```
  <endTime>23:59</endTime>
  <repeat>
    <dayOfMonth>*</dayOfMonth>
    <month>*</month>
    <dayOfWeek>?</dayOfWeek>
    <year>*</year>
  </repeat>
  <enabled>true</enabled>
</schedule>
```

Possibly the best feature of the schedule definition is that most of it contains defaults, so you only need to define a start date and if it's enabled, and you will have a schedule that operates forever from the start date, 24 hours a day.

```
<schedule>
  <startDate>2011-01-01T00:00:00Z</startDate>
  <enabled>true</enabled>
</schedule>
```

State

State is an additional layer of configuration that helps determine how to interpret the schedule. As noted above, the enabled field indicates whether the service is active or not. The state field indicates whether the time/date and cron values evaluate from a positive or negative perspective.

Example: state == ON, date range Jan 2000 - Jan 2001, time range = 9AM-5PM

```
<schedule>
  <startDate>2000-01-01T00:00:00Z</startDate>
  <endDate>2001-01-31T00:00:00Z</endDate>
  <state>ON</state>
  <startTime>09:00</startTime>
  <endTime>17:00</endTime>
  <enabled>true</enabled>
</schedule>
```

The above evaluates as: if at the time of evaluation the date is within range and the time is within the time range, the Policy Engine will return serviceActive=true and the Policy Engine will turn the service on or keep it on if already started for the subscriber.

Example: state == OFF, date range Jan 2000 - Jan 2001, time range = 9AM-5PM

The above evaluates as: if at the time of evaluation the date is within range and the time is within range, the Policy Engine will return serviceActive=false and the Policy Engine will turn the service off.

Custom Search Params

The basic search provides for name and credential matching. However, there is a large data structure that can be matched against. The data objects are structured as a large HashMap or KEY:VALUE pairs. Lists of HashMaps can be included as well. An industry-standard term for KEY:VALUE pairs is Attribute:Value pairs or AVPs. That is why the SearchSubscribersRequest uses "avp" as the element tag for complex search parameters. Currently, the Search views all parameters as an AND operation.

Each avp contains 2 children: code and value. The Code represents the KEY in the data HashMap, and the Value is the VALUE associated to that KEY in the data HashMap. For example, the following code block represented in JSON (JavaScript Object Notation) shows some of the key data points of a Subscriber.

We have tried to be consistent and append "_key" to the KEY portion of the KEY:VALUE pair. Notice that version_key is the KEY for an integer value, and that services_key is the KEY for a List of "services". Each service is also a HashMap of KEY:VALUE pairs.

Note **NoSQL**

It's important to note that the CPS database is a NoSQL database. It does not use tables and columns to structure the data. Because each record is a "document" (which is a HashMap of HashMaps), you can access keys in the same way you access properties in JavaScript with dot notation. In the example below, if you want to find subscribers who have the service AVP whose code is AVP_CODE use **services_key.avps_key.code_key**.

```
{ "_id_key" : null,
    "version_key" : 0,
    "services_key" : [
        {
            "code_key" : "GOLD",
            "enabled_key" : true,
            "avps_key" : [
                {
                    "code_key" : "AVP_CODE",
                    "value_key" : "AVP_VALUE"
                },
                {
                    "code_key" : "AVP_CODE_2",
                    "value_key" : "AVP_VALUE"
                }
            ]
        }
    ],
    "name_key" : { "full_name_key" : [ "Test", "Subscriber" ] },
    "status_key" : "ACTIVE",
    "credentials_key" : [
        {
            "network_id_key" : "networkId1",
            "password_key" : "password",
            "expiration_date_key" : null
        }
    ],
    "avps_key" : [
        {
            "code_key" : "SUBSCRIBER_AVP_CODE",
            "value_key" : "SUBSCRIBER_AVP_VALUE",
        }
    ]
    }
```

Example: Services:GOLD and Rate Plan:6MO_DISCOUNT. This should return all users who have the Gold Service and the 6 month discounted payment plan.

```
<se:Envelope xmlns:se="http://schemas.xmlsoap.org/soap/envelope/">
  <se:Body>
    <SearchSubscribersRequest xmlns="http://broadhop.com/unifiedapi/soap/types">
      <filter>
        <avp>
          <code>services_key.code_key</code>
          <value>GOLD</value>
        </avp>
        <avp>
          <code>billing_info_key.rate_plan_code_key</code>
          <value>6MO_DISCOUNT</value>
        </avp>
      </filter>
    </SearchSubscribersRequest>
```

```
  </se:Body>
</se:Envelope>
```

Example: AVP:UPLINK. This should return all users who have an AVP with code = uplink.

```
<se:Envelope xmlns:se="http://schemas.xmlsoap.org/soap/envelope/">
  <se:Body>
    <SearchSubscribersRequest xmlns="http://broadhop.com/unifiedapi/soap/types">
      <filter>
        <avp>
          <code>avps_key.code_key</code>
          <value>SUBSCRIBER_AVP_CODE</value>
        </avp>
      </filter>
    </SearchSubscribersRequest>
  </se:Body>
</se:Envelope>
```

Balance Engine Thresholds

The Threshold table in Policy Builder defines thresholds that trigger messages when quota is credited/debited and therefore qualifies as breached/unbreached. These messages are sent back to the Policy Engine from MSBM on Credit, Debit, Charge, and Provision functions so that a policy can make decisions and take actions based on the threshold breach. When breached, the current amount is reported.

Thresholds can be defined for an Account Balance Template (monitors all child quotas as an aggregate) and for a Quota Template (only monitors the credits of that quota). Thresholds operate against the total of all currently valid credits under the specified balance/quota. A currently valid credit is a credit for which the start date is before the current date/time and the end date is after the current date/time.

Note **Threshold Calculation**

Thresholds are based on charged amounts. Reserved amounts are not included.

Example

- A Subscriber has a credit of 1 GB that ends on Oct 15th
- The system is configured with a Percentage threshold of 90% on a Balance template
- The Subscriber uses 922 MB of credit
- The threshold has been crossed: 922 MB is 90% of 1024 MB (1 GB)
- The Subscriber purchases more credit that ends on Oct 30th - another 1 GB is credited
- The threshold is recalculated and is now at 45%. The calculation formula is: ((amount charged / the original credit amount) * 100) In this case, (922 MB / 2048 MB) * 100
- However, once the date passes Oct 15th, the first credit expires and is no longer used in the threshold calculation. As a result, if the Subscriber has not used any more quota, the threshold will be calculated at 0% - (0 MB / 1024) * 100

Note **Determing The Calculation Values**

The original amount that a threshold is compared against can be determined using the sum of balanceTotal + debitedTotal + reservedTotal returned in a QueryBalance request. The amount charged is the debitedTotal. Percentage thresholds are calculated as (debitedAmount/(balanceTotal + debitedTotal + reservedTotal)) * 100.

Reference Data vs. Subscriber Specific

Reference Data thresholds (RDT) are defined on the Balance or Quota Template to which they apply in Policy Builder. They are system or global thresholds and are applied to all subscribers who have purchased the related Balance or Quota package.

Subscriber Specific thresholds (SST) are defined via API or Policy Action. SSTs are only applicable for the subscriber for which the SST was defined. You must defined the SST individually for each subscriber for whom you want the threshold to apply.

Note **Unique Names**

Thresholds must have unique names. SSTs and RDTs must have unique names as well. You can use the same SST code name for multiple subscribers, but that value must be unique compared to the name values for the RDTs.

Reduction of Reservation Granted Amounts

A threshold defined on an Account Balance Template does reduce the reservation amount as it nears the threshold.

A threshold defined on a Quota Template does NOT reduce the reservation amount as it nears the threshold.

When the reservation granted amount is reduced from the requested amount due to a threshold, the quota granted is reduced to the amount between the current usage level and the value where the threshold would be breached. This reduction continues on each successive reservation until the Default Minimum Dosage defined on the Balance Plugin Configuration is reached. After that value is reached for the granted amount, the next reservation will go back to normal behavior and trigger the breach occurred condition.

Soft Thresholds

All thresholds in CPS are soft thresholds.

A soft threshold allows the Balance Engine to grant the minimum dosage even though it could cause the subscriber's balance to breach a threshold.

Threshold Groups

It is possible to group thresholds so that they operate in concert. By adding a group name in the Policy Builder configuration, thresholds in the same group are evaluated in the order they appear in the table (top to bottom). The Balance Engine will then only send notifications to the Policy Engine for the first threshold breach found.

Example

- The system is configured with three Percentage thresholds on a Balance template: 80%, 60% and 50% in descending order
- All three thresholds are grouped, for example, CPSPercents
- When a Subscriber's usage gets to 62%, the Balance Engine will only send notifications for the 60% threshold
- Once the Subscriber's usage goes above 80%, the Balance Engine will only send notifications for the 80% threshold and will not send notifications for the 60% or 50%

Note **Threshold Group Order**

Threshold group processing is based on the actual order of the thresholds in the Policy Builder configuration table NOT on the highest value. For example, if there are 2 thresholds: 60% and 80% and the 60% threshold is the top or first one listed, then notifications for the 80% threshold will never get sent. However, if the thresholds are defined as amount remaining instead of amount used (amount used is the default), then notifications for the 80% threshold will get processed and sent.

Bill Cycle

Recurring Quota now has the concept of Bill Cycle. If you decide to use a Bill Cycle Quota, it supercedes the use of manually setting Recurring Refresh dates. See Last Recurring Refresh (LRR) for more information.

Note **Converting Recurring Quota to Bill Cycle**

It is possible to change a Recurring Quota template to use Bill Cycle. For an existing subscriber, the LRR date is used going forward but the recurrence frequency and all other behaviors for the quota are then based on the Bill Cycle definition and are implemented during the next refresh. Existing subscribers cannot have 29, 30, or 31 as their Bill Cycle refresh date using a converted quota.

Bill Cycle values are 1 - 31 inclusive.

29th, 30th, and 31st

Compared to LRR, Bill Cycle handles the end of month problem in a much more intuitive manner. If the LRR is set to the 29th, 30th, or 31st with a Recurring Quota, the LRR gets changed to 28 on the next refresh. Whereas, if the bill cycle day for a Bill Cycle Quota is set to 30 the refresh in February, for example, will be on the 28th or on the 29th in a leap year. However, in any other month, the refresh will happen on the 30th as expected.

Note **Updating Bill Cycle**

Bill cycle can be changed by referring to *ChangeBillCycleRequest* API.

Note **ChangeRecurringRefreshDay API**

Do not use the ChangeRecurringRefreshDay API with Bill Cycle Quotas, instead use *ChangeBillCycleRequest*.

Note **Recurrence Frequency Amount**

For a regular Recurring Quota, the Recurrence Frequency Amount (RFA) field adds an additional layer of control to when the quota refreshes. If the RFA is set to 2, then refresh will wait 2 periods before refreshing the quota. This way you could have quota refresh every 2 months instead of every month. For Bill Cycle Quota, the Recurrence Frequency Amount (RFA) is ignored. Refresh happens once every bill cycle.

Last Recurring Refresh (LRR)

Recurring Quota uses the concept of Last Recurring Refresh (LRR) to properly calculate the refresh of the recurring quota.

LRR is the date that a recurring quota was provisioned or last refreshed by the Balance engine. It is the date that the system uses to determine the next time a recurring quota should be refreshed.

Note **Monthly Recurring Quota**

Be particularly careful setting this manually with a recurring quota if the refresh frequency is set to monthly. A month is not 30 days. When set to monthly, an actual month is used for the calculations and this does vary depending on which month you are doing this in as well as other factors like Daylight Savings Time.

Refreshed means a new credit will be created automatically on the next Balance action after the LRR + template recurrence frequency (the value of this calculation equals the Next Refresh Date). For example, if a recurring quota is defined as monthly and the LRR is "Wed Mar 28 2012 15:05:11 GMT-0600 (MDT)" (typically this will also be the start date of the corresponding credit), then the next refresh will occur on or after "Fri Apr 27 2012 15:05:11 GMT-0600 (MDT)" which should typically be the end date of the corresponding credit.

The refresh occurs on the next Balance action instead of on the actual next refresh date so that not all subscriber accounts refresh at the exact same moment, thus balancing load and resources. However, it should be noted that the date of the new credit created by the refresh will still have its dates based on the actual stored LRR and not on when it is actually refreshed by the Balance engine. The new credit will have a start date equal to the new LRR after the refresh has occurred. The new credit end date will be the start date + recurrence frequency. This value is also the new Next Refresh Date.

The LRR can be overridden in the CreateBalance or ChangeRecurringRefreshDay or ChangeBillCycle APIs.

Note **Override LRR**

If you override LRR, make sure that the start date and end date align properly. To do this consistently, set the startDate value the same as the LRR. This will ensure that the endDate equals the next refresh date (LRR + template recurrence frequency) so that the provisioned credit ends when the refresh (new credit is created) occurs.

29th, 30th, and 31st

- If the LRR is set to the 29th, 30th, or 31st, it will remain at the date until the first refresh and/or rollover event, then the LRR will be set to the 28th.
- Customers should be encouraged to not use dates of the 29th, 30th, or 31st, particularly if this is tied to their billing.
- Customer should be informed that while they can provision on the 29th, 30th, or 31st, the refresh date will "float" to the 28th the next month.

Note **30 days vs 1 month**

If you use a number of days, for example 30, instead of 1 month, the Balance engine will refresh on the exact number of days, but that will cause the refresh date to "float" to a different day of the month every month since no two consecutive months have the same number of days (except July and August).

Id, ParentId and Version

These fields require special handling. Do not modify id, parentId, and version at all for any reason.

These fields are marked as optional in the schema because the Unified API re-uses objects - in particular the subscriber object in creates and updates. If the id, for example, which represents the database generated id value was a required field, then the CreateSubscriber call would require a value. This does not make sense since the subscriber object is not yet in the databse.

The version field is used for optimistic locking, since MongoDB does not implement it. Optimistic locking is the concept of managing concurrent updates to objects using a value that increments in a known way for each modification. If the version field does not match the expected value on update, it is assumed that another thread modified the object and therefore the data is now "dirty".

XSS - Cross Site Scripting Defense

Each incoming request is now checked for dangerous characters and code.

Two regexes are used to check each request:

- ^.*?(?:[^\\p{L}\\p{Nd}\\p{Nl}!@#-_/:\'"\\s={}\\+]|[$^()\\\\]|([&]
  (?!(amp|apos|gt|lt|quot)[;]))|(?<!([&](amp|apos|gt|lt|quot)))[;])+.*?$
- <!\\[CDATA\\[.*?[<>].*?(?!\\]\\]>)

The first regex returns true if the request contains any characters that are not word characters, !@#-_/:'" or white space or if the request contains any of these characters $^&();\ The first regex allows & and ; if they are part of the XML 1.0 valid entities (amp,apos,gt,lt,quot). The second regex checks if <> are inside CDATA tags. Another way to explain the two regexes is that the following characters are allowed: alphanumeric including unicode for other languages, white space, valid XML 1.0 entities, .!@#-_/:'"?*[]={}+,% and <> except when inside CDATA tags.

For example, this is a valid request:

<CreateSubscriberRequest><subscriber><credential><networkId><![CDATA[testcredential]]></networkId></credential><service><code>GID</code><enabled>true</enabled></service><status>ACTIVE</status></subscriber></CreateSubscriberRequest>

This is an invalid request:

```
<CreateSubscriberRequest><subscriber><credential><networkId><![CDATA[<script
alert(document.cookie);>]]></networkId></credential><service><code>GID</code><enabled>true</enabled></service><status>ACTIVE</status></subscriber></CreateSubscriberRequest>
```

See CDATA for more information about using CDATA tags and XML entities with the Unified API.

Note **Data Compatibility**

Please note that because of the XSS restriction in the API, a deployment should only use the allowed character set for all configuration in Policy Builder to make sure that all data is compatible. It is possible to adjust the regex and to determine if only cdata is checked via two properties: -Dua.xss.pattern=#REGEX_PATTERN# and -Dua.xss.check.cdata.only=false. If no pattern is used: -Dua.xss.pattern="", then ua.xss.check.cdata.only will be set to true.

Note **Avoid**

Even though **$ & <** are the only restricted characters in the Control Center and Unified API from a schema perspective, considering the XSS checks, at a minimum, it is best to avoid the following characters: **$^&();=+<** See CDATA for more information about using CDATA tags and XML entities with the Unified API.

Error Codes

The **%s** is used as a replacement value so that more meaningful information can be included in the message.

Note **Error Codes**

Please note that due to API changes and bug fixes, some of the error codes are no longer used.

Note **Error Codes**

- Codes 9 and below apply to all APIs.
- Codes 10-15 are the Subscriber APIs like CreateSubscriber, DeleteSubscriber, etc.
- Codes 17-19 apply to all APIs.
- Code 55 is specifically related to password hashing for all APIs that modify credentials.
- For all other codes - the names match the request.

code	name	message
0	SUCCESS_CODE_GENERIC	Request completed successfully
1	SUCCESS_CODE_VALIDATION	Validation completed successfully
2	ERROR_CODE_GENERIC	Unable to process the request
3	ERROR_CODE_NULL	Object: %s is null
4	ERROR_CODE_ILLEGAL_VALUE	Invalid XML: %s
5	ERROR_CODE_ILLEGAL_VALUE	Illegal Value: %s
6	ERROR_CODE_INVALID_REQUEST	Invalid Request: %s
7	ERROR_CODE_INVALID_RESPONSE	Invalid Response: %s
8	ERROR_CODE_REQUIRED_DATA	Required Data: %s
9	ERROR_CODE_NON_UNIQUE	Duplicate Value for Unique Data Constraint: %s
10	ERROR_CODE_CREATE	Error Creating Object: %s
11	ERROR_CODE_UPDATE	Error Updating Object: %s
12	ERROR_CODE_UPDATE_VERSION	Optimistic Locking Error - the version number does not match the database version, another party has probably updated the data. Refresh the request data and try the request again
13	ERROR_CODE_DELETE	Error Deleting Object: %s
14	ERROR_CODE_DELETE_ CREDENTIAL_BALANCE_ID	Error Deleting Credential: The networkId(s) [%s] match(es) a balance id - please change the balance id before deleting the credential(s).

code	name	message
15	ERROR_CODE_SEARCH	Error Searching for Object with key: %s
16	ERROR_CODE_AUTHENTICATE	Error Authenticating User/Subscriber Object with credential: %s
17	ERROR_CODE_SERVLET_EXCEPTION	Servlet Processing Error: %s
18	ERROR_CODE_WS_MODULE_ NOT_INSTALLED_EXCEPTION	The expected module is not installed: %s
19	ERROR_CODE_WS_API_NOT_ IMPLEMENTED_EXCEPTION	The requested api is not implemented at this time
20	ERROR_CODE_QUERY_SESSION	Error Querying Sessions(s) for Subscriber: %s
21	ERROR_CODE_REFRESH_SESSION	Error Refreshing Subscriber Profile: %s
22	ERROR_CODE_START_SESSION	Error Starting Session(s) for Subscriber: %s
23	ERROR_CODE_STOP_SESSION	Error Stopping Session(s) for Subscriber: %s
24	ERROR_CODE_UPDATE_SESSION	Error Updating Session for Subscriber: %s
25	ERROR_CODE_CREATE_BALANCE	Error Creating Balance for Subscriber: %s
26	ERROR_CODE_CREDIT	Error Crediting Quota for Subscriber: %s
27	ERROR_CODE_DEBIT	Error Debiting Quota for Subscriber: %s
28	ERROR_CODE_DELETE_BALANCE	Error Deleting Balance for Subscriber: %s
29	ERROR_CODE_DELETE_QUOTA	Error Deleting Quota for Subscriber: %s
30	ERROR_CODE_QUERY_BALANCE	Error Querying Balance for Subscriber: %s
31	ERROR_CODE_ROLLOVER_CREDIT	Error Rolling Over Credit for Subscriber: %s

code	name	message
32	ERROR_CODE_UPDATE_BALANCE	Error Updating Balance for Subscriber: %s
33	ERROR_CODE_CREATE_VOUCHER	Error Creating Voucher: %s
34	ERROR_CODE_DELETE_VOUCHER	Error Deleting Voucher: %s
35	ERROR_CODE_QUERY_VOUCHER	Error Querying Voucher: %s
36	ERROR_CODE_EXECUTE_ACTION	Error Executing Action: %s
37	ERROR_CODE_DELETE_CREDIT	Error Delete Credit for Subscriber: %s
38	ERROR_CODE_AUDIT	Error Auditing: %s
39	ERROR_CODE_PURGE_ AUDIT_HISTORY	Error Purging Audit History: %s
40	ERROR_CODE_QUERY_ AUDIT_HISTORY	Error Querying Audit History: %s
41	ERROR_CODE_AUDIT_ MGR_IS_NOT_ENABLED	The Audit Module is not enabled. Please check the plug-in configuration.
42	ERROR_CODE_GET_ SUBSCRIBER_COUNT	Error Getting the Subscriber Count: %s
43	ERROR_CODE_ GENERATE_BATCH	Error Generating the Voucher Batch: %s
44	ERROR_CODE_ REDEEM_VOUCHER	Error Redeeming the Voucher for Subscriber: %s
45	ERROR_CODE_ CHANGE_STATUS	Error Changing the Status for Subscriber: %s
46	ERROR_CODE_ CHANGE_SUBSCRIBER_AVPS	Error Changing the Avps for Subscriber: %s
47	ERROR_CODE_ UPDATE_SERVICE	Error Updating the Service for Subscriber: %s
48	ERROR_CODE_ ADD_SERVICE	Error Adding the Service for Subscriber: %s

code	name	message
49	ERROR_CODE_ DELETE_SERVICE	Error Deleting the Service for Subscriber: %s
50	Error Deleting the Service for Subscriber: %s	Error Deleting the Voucher Batch: %s
51	ERROR_CODE_SWITCH_SERVER	Error Switching the Service for Subscriber: %s
52	ERROR_CODE_EXTEND_CREDIT	Error Extending the Credit for Subscriber: %s
53	ERROR_CODE_ GET_REF_DATA_SERVICES	Error Getting Service Reference Data: %s
54	ERROR_CODE_ GET_REF_DATA_BALANCE	Error Getting Balance Reference Data: %s
55	ERROR_CODE_ENCRYPTION	Error Encrypting: %s
56	ERROR_CODE_ADD_SSID	Error Adding SSID: %s
57	ERROR_CODE_UPDATE_SSID	Error Updating SSID: %s
58	ERROR_CODE_DELETE_SSID	Error Deleting SSID: %s

Policy Engine Error Codes

The Execute Action, Query Session, and Stop Session APIs send requests into the Policy Engine and interact with the policy state. The Policy Engine has a set of error codes that can be returned in the error messages that get returned by the APIs.

Table 2: Policy Engine Error Codes

Code	Message	Note
SS002	• Avps are empty • Port and/or ISG IP Address AVPs are empty	-
SS003	-	Failed to get a response object from startSession
SS004	-	login error
SS005	-	timeout (usually a COA timeout)
QND001	Avps are empty	Query Network Device
AR00	Success	-

Code	Message	Note
AR01	Success ALLOW_ALL authorization	-
AR02	Failed USUM_AUTHORIZATION no domain found	-
AR03	Failed USUM_AUTHORIZATION no user id retriever	-
AR04	Failed USUM_AUTHORIZATION no user id found	-
AR05	Failed USUM_AUTHORIZATION no password found for user	-
AR06	Success USUM_AUTHORIZATION	-
AR07	Failed USUM_AUTHORIZATION password and/or user name do not match	-
AR08	Failed USUM_ONLY_AUTHORIZATION no domain found	-
AR09	Failed USUM_ONLY_AUTHORIZATION no user id retriever	-
AR10	Failed USUM_ONLY_AUTHORIZATION no user id found	-
AR11	Failed USUM_ONLY_AUTHORIZATION no password found for user	-
AR12	Success USUM_ONLY_AUTHORIZATION	-
AR13	Failed USUM_ONLY_AUTHORIZATION password and/or user name do not match	-
AR14	Success ANONYMOUS_AUTHORIZATION - user id, password match	-
AR15	Failed ANONYMOUS_AUTHORIZATION - user id matches, password does not match	-

Code	Message	Note
AR16	Success ANONYMOUS_AUTHORIZATION - user id matches - no password check	-
AR17	Failed ANONYMOUS_AUTHORIZATION - user id does not match anonymous user id	-
AR18	Failed ANONYMOUS_AUTHORIZATION - no user id retriever	-
AR19	TAL success	Existing Subscriber
AR20	TAL success	TAL with no domain
AR23	AAA_AUTHORIZATION success	-
AR24	AAA_AUTHORIZATION success due to timeout	-
AR25	Authorization failed for the following user ['$userName'] to server ['$proxyAAAAuthorization. getAaaServer().getName()']	Access Reject Message
AR26	Could not find a User ID from this message using the retriever: '$userIdRetrieverClassName'	-
AR27	User ID '$userId', does not equal one-click User ID: '$oneClickVoucher. getOneClickUserId()'	-
AR28	one-click-voucher success	-
AR29	Password provided: '$password' does not equal one click password: '$oneClickVoucher. getOneClickPassword()'	-
AR30	Voucher is expired	-
AR31	Voucher authenticated	-

CDATA

The Unified API can accept CDATA tags for all fields.

Use the Plugin configuration in Policy Builder to set which fields will get CDATA tags for outgoing responses. By default, the following fields will have CDATA tags in responses: networkId, password, data, oldNetworkId, oldPassword, newPassword.

Note **Policy Builder Plugin CDATA Configuration**

Make sure to remove spaces from the CDATA fields configuration: networkId,password,data,oldNetworkId,oldPassword,newPassword

XML Entities

The Unified API can accept CDATA tags for all fields. If a field has a CDATA tag, any XML entities will not get resolved.

For example, the database will store the literal characters **& <**, if the following is sent in a request:

<SomeUnifiedApiRequest> ... <someElement><![CDATA[& <]]></someElement> ... </SomeUnifiedApiRequest>

Conversely, the API will resolve the entities and the database will store **& <** if the following is sent in a request:

<SomeUnifiedApiRequest> ... <someElement>& <</someElement> ... </SomeUnifiedApiRequest>

Note **Resolved XML Entities** The Unified API only resolves XML entities for requests and does not resolve stored data back into XML entities in responses! Therefore in the above example, where the **& <** literal characters were stored in the database; those characters will now make the response invalid according to the XML 1.0 specification. Therefore, CDATA must be used when sending XML entity references in requests. Invalid response: <SomeUnifiedApiRequest> ... <someElement>& <</someElement> ... </SomeUnifiedApiRequest>

Namespace

Target Namespace	http://broadhop.com/unifiedapi/soap/types
Element and Attribute Namespaces	• Global element and attribute declarations belong to this schema's target namespace. • By default, local element declarations belong to this schema's target namespace. • By default, local attribute declarations have no namespace.