HCLSoftware

HCL Detect v12.1.8 Development Guide

Contents

Chapter 1. Folder Structure

A basic solution folder structure mainly consist of the following folders mentioned below. The structure is as follows:

```
.
├── acme
│   ├── etc
│   ├── jupyter
│   ├── lib
│   ├── python
│   └── web
├── drive
│   ├── bin
│   ├── doc
│   ├── etc
│   ├── include
│   ├── lib
│   └── python
├── external
│   ├── apache-tomcat-9.0.86
│   ├── boost-1.84.0-python-3.11.9
│   ├── gcc-8.2.0
│   ├── kafka_2.12-3.7.0
│   ├── pyenv-3.11.9
│   ├── redis-5.0.5
│   └── server-jre-1.8.0_202
├── HCLDetectv12.1.8-LICENSE.txt
└── HCLDetectv12.1.8-NOTICE.txt
```

Let's go through each folder and understand the contents within it.

`acme` folder

The `acme` folder is solution folder, which mainly contains the information specific to the customer code.

`bld` folder

The `bld` folder mainly contains the generated or compiled code. It also contains the installation files which one would find in an installer tarball. The structure is as follows:

```
$ tree -L 1 /home/user/stash/goliath/bld/
/home/user/stash/goliath/bld/
├── current_arch -> /home/user/stash/goliath/bld/rhel07
├── external_dependencies -> /home/user/HCL/goliath/1.0.0/rhel07
└── rhel07
```

Based on your running operation system (`rhel07` / `rhel08`) a folder would be created. Some of the important paths are:

> “

> - **`HCL_HOME` = /home/user/stash/goliath/bld/rhel07/install/goliath/drive/**
>   - This folder contains the compiled code, executable and configuration files for the product Detect.
> - **`HCL_INSTANCE_HOME` = /home/user/stash/goliath/bld/rhel07/instance_home/goliath**
>   - This folder contains the runtime generated files and logs. This also contains the flat files where NameService, Kafka, PinPoint and FastPast store their data.
>   - Also, for feed applications which use files for input data, the default input file path would be here.
>
> ”

`dev_env_bootstrapper` folder

This folder contains the scripts which sets up your bash environment, Python environment, environment variables and other environment settings required to build, develop and bring up the services.

This also contains a `Makefile` which is referred when any make command is executed.

`drive` folder

This folder contains the built product code which the solution would be referring to.

`etc` folder

This folder contains solution based configuration files. This will explained in more detail in the :ref:`configuration<solutionconfiguration>` section.

`infra` folder

This folder contains the built infra code which the solution would be referring to. Infra contains some build related code, helper classes and methods and commonly used utilities such as database connectors, logger methods, etc.

`Makefile` file

This is actually a symlink which points to a Makefile within the `dev_env_bootstrapper` folder.

`pom.xml` file

Parent pom file for the solution which is required as part of the Java build of the solution. This file would contain the Maven dependencies that the solution imports and build plugins required.

`src` folder

Contains solution code for the feed applications and helper functions.

`test` folder

Contains test code for running automated test on the solution's feed applications.

Chapter 2. Drive.JSON File

The drive.json file stores essential configurations for the HCL Detect application, including database details, time zone, location, metrics, admin credentials, and more. Before running the application, users must configure these settings or the application will consider the default values provided in the defaultdrive.json file.

Let us go through each parameters in the drive.json file.

Communication Services

`communicationServices`:

Communication services are configured to communicate with the Kafka based services. For communication services and communication configuration, the values are pre-configured and should not be modified.

Drive

`administratorUser`:

This section allows configuring the admin user profile.

```
"administratorUser": {
            "displayName": "Administrator",
            "firstName": "Administrator",
            "lastName": "User",
            "username": "admin"
        },
```

Configure the initial administrative user account for HCL Detect. Upon first application startup, Tomcat creates an admin user based on these settings. These details cannot be modified after initial setup.

Note: It is mandatory to configure the default user profile before the initial launch of the HCL Detect application.

`applicationHealthMetricsRefreshPeriodInSeconds`:

Set your preferred refresh time for the System Health page in seconds.

`applicationPartitionStartupTimeoutInSeconds`:

Configure the start-up timeout for the feed application partition.

`audience`:

Configure the audience criteria parameters like timeouts, query limit, bucket size for chart, and reporting options. For more information, refer Audience Criteria on page 9.

`auditLogConfiguration`:

```
        "auditLogConfiguration": {
            "maxFileSizeInMBs": 1024,
            "maxNumFiles": 5
        },
```

`campaignActuator`:

Configure the core components of the Campaign Actuator. For more information. refer Campaign Actuator on page 10.

`communicationConfiguration`:

Communication services are configured to communicate with the Kafka based services. For communication services and communication configuration, the values are pre-configured and should not be modified.

`currency`:

Specify the default currency code and symbol to be displayed in transactional data. This setting determines the monetary format for all financial transactions within the system. Currently, only single monetary format is supported.

`database`:

Configure the database details including, name, server address, port number, database type, and credentials.

```
"database": {
            "name": "drive_acme_core",
            "server": {
                "ip": "127.0.0.1",
                "port": 3306
            },
            "type": "MariaDB",
            "userName": "drive",
            "userPassword": "6F2FA7F6B3CD1570F19F0A0B2636E033"
        }
```

`name`: Include the database name.

`server`: Configure the server address and port number of database.

`type`: Set the database type as *MariaDB*.

`userName`: Enter the user name for the database.

`userPassword`: Set the encrypted password. HCL Detect uses internal password encryption tool to encrypt the passwords.

`demoModeEnabled`, `displayLocale` and `distanceDisplayUnitSystem`:

These fields are currently inactive and will be utilized for future enhancements.

`driveUser`:

Define the credentials for a dedicated database user account that Tomcat will create to facilitate communication with other applications. Tomcat will cache the drive user details from the drive.json file after the initial startup.

Note: Don't change the drive user profile after the initial startup of Tomcat server.

```
"driveUser": {
            "displayName": "Drive User",
```

```
            "email": "support@hcl.com",
            "firstName": "Drive",
            "lastName": "User",
            "password":
"97A08DFA3379B540B1BF196628B990F866F71A8E98531B26430E845F749478DE5C2E76453040705A066DC771DA41E7F7"
        },
```

`"displayName": "Drive User", "email": "support@hcl.com", "firstName": "Drive", "lastName": "User",`: configure the display name, email address, first and last names of drive user profile.::

`password`: an encrypted password is automatically generated and stored within the drive.json file during the installation process.

`enabledFeatures`:

Don't modify the default values. These fields be utilized for future enhancements.

`eventEndpoints`:

Don't modify the default values. These fields be utilized for future enhancements.

`externalKafkaConfiguration`:

This field is currently reserved for future enhancements.

`fastPast`:

Specify the number of partitions required for FastPast. You can either provide specific port numbers or simply indicate the desired number of partitions, allowing the system to automatically assign ports.

```
"fastPast": { "ports": [ 41235 ] }
```

`"ports": [ 41235 ]`: Configure the specific port numbers for FastPast partitions.

`"numServers": [ 2 ]`: Set the number of servers for FastPast partitions that can run on system assigned ports.

`feedApplications`:

Configure feed applications and their corresponding parameters. For more information, refer Feed Applications on page 11.

`feedDataModels`:

Feed data models represent common classes of feed data sources in the system. For more information about Feed data models, refer Feed Data Models on page 19.

`fulfillmentServices` :

Don't modify the default value. These objects are utilized for future enhancements.

`internalKafkaConfiguration`:

Specify the port number for Kafka port and zoo keeper port. If not specified, the system will assign random port number.

`jupyterNotebookPort`:

This field is reserved for future implementation.

`mapCenter`:

Configure the latitude and longitude positions to define the center position of the map. Currently, this field is reserved for future enhancements.

`pinPoint`:

Specify the number of partitions required for PinPoint. You can either provide specific port numbers or simply indicate the desired number of partitions, allowing the system to automatically assign ports.

```
"pinPoint": {
            "ports": [ 41233 ]
        }
```

`"ports": [ 41235 ]`: Configure the specific port numbers for Pinpoint partitions.

`"numServers": [ 2 ]`: Set the number of servers for Pinpoint partitions that can run on system assigned ports.

`profiles`:

Configure the profiles set in the feed applications. For more information about profiles, refer Profile on page 15.

`referenceDatasets`:

Configure reference datasets to populate the profile tables. This defines the data model and format of reference data along with the destination profile and profile table. For more information about reference datasets, refer Reference Datasets on page 16.

`tomcatConfiguration`:

Configure tomcat parameters. For more information, refer Tomcat Configuration on page 14.

Event Consumers

`eventConsumers`:

Set the Unica Interact parameters to send the event details. For more information about event consumers, refer to Integrations on page 49.

Fulfillment Services

`fulfillmentServices`:

Don’t modify the default value. These objects are utilized for future enhancements.

Running Mode

`runningMode`:

Set the mode as "*notDisturbuted*" for VM only mode, and "*scaledDistributed*" for hybrid mode.

Solution

`solution`:

Specify a unique customer name. The `mainClass` field is pre-configured, and should not be modified.

Audience Criteria

This section explains about configuring the parameters like timeouts, query limit, bucket size for chart, and reporting options for audience criteria.

```
"audience": {
            "clientInactivityTimeoutInMillis": 600000,
            "maxNumOfActiveQueriesCached": 10,
            "maxNumOfBucketsForSparseNumericalHistograms": 10000,
            "maxNumOfConcurrentQueriesUnderExecution": 10,
            "maxNumOfEvaluationErrorsToReport": 10,
            "maxNumOfUniqueValuesForCategoricalHistograms": 200000,
            "queryExpirationTimeoutInMillis": 3600000,
            "queryFilterSamplingRate": 1.0
        },
```

`clientInactivityTimeoutInMillis`: set the maximum idle time allowed before a client is considered inactive. During the event execution, when the browser is idle for the set time period, the system will throw timeout message.

`maxNumOfActiveQueriesCached`: set the maximum number of active queries that can be cached.

`maxNumOfBucketsForSparseNumericalHistograms`: set the maximum number of buckets that can be generated in the numerical histogram chart.

`maxNumOfConcurrentQueriesUnderExecution`: set the maximum number of concurrent queries that can executed at once. Let assume, the set number of concurrent queries be three, system will execute the first three queries, and execute the fourth query when one of the three queries get completed.

`maxNumOfEvaluationErrorsToReport`: set the maximum number of evaluation errors to be listed in the report following audience criteria evaluation. The report contains error details of the condition and warnings.

`maxNumOfUniqueValuesForCategoricalHistograms`: set the maximum number of segments or categories that can be generated from unique values in a histogram chart.

`queryExpirationTimeoutInMillis`: set the query execution timeout threshold in milliseconds.

`queryFilterSamplingRate`: set a filter sampling rate within the range of 0 to 1 to sample the records from each partitions and execute the audience criteria. For example, if you set the sampling rate as 0.5, the system will sample half the records from each partition and perform the audience criteria evaluation in each partition.

Campaign Actuator

This section allows configuring the core components of the Campaign Actuator, including data ingestion from multiple feeds, data processing based on triggers, event execution aligned with contact policies.

Note: Restart the campaign actuator, if the initial configuration is modified or altered.

```
"campaignActuator": {
            "contactPolicyCheckerSettings": {
                "batchSize": 1,
                "perUserContactPolicy": {
                    "countBasedPolicies": [
                        {
                            "countThreshold": 50,
                            "timeUnit": "Days",
                            "windowSize": 7
                        }
                    ]
                }
            },
            "deduplicatorFsyncBatchSize": 1000,
            "feedDataKafkaSourceSettings": {
                "batchSize": 1000
            },
            "logLevel": "INFO",
            "mergerSettings": {
                "inputBufferSize": 1000,
                "maxBlockingTimeInMillis": 500
            },
            "name": "Campaign Actuator",
            "numParallelChannels": 2,
            "responseMessageKafkaSourceSettings": {
                "batchSize": 1000
            },
            "triggerEvaluatorSettings": {
                "batchSize": 1,
                "maxAllowedTimeLagInSeconds": 86400,
                "stateExpirationCheckIntervalInMillis": 5000
            },
            "useKafkaToProcessActuation": false
        },
```

contactPolicyCheckerSettings

Configure the batch size and control the number of events to be sent to a user within a specific period.

`batchSize`: define the batch size of the events to be executed based on the configuration in the Contact policy. Preferred batch size for production environment is "100", and for test environment, it should be "one".

`"countBasedPolicies": [ { "countThreshold": 50, "timeUnit": "Days", "windowSize": 7}]`: in the perUserContactPolicy section, you can set the maximum number of campaign events that can be sent to a user in a specific time interval. The ***timeUnit*** can be ***Days*** or ***Months***. In the above example, the system will send up to 50 events for a specific user within seven days.

`deduplicatorFsyncBatchSize`: set the number of records to be processed in a single batch during data synchronization to avoid duplicate data sync up.
`feedDataKafkaSourceSettings`: set the number of records to be read by campaign actuator from feed applications.
`logLevel`: configure the level of log information required from the campaign actuator execution. You can set any one of the following options:

- ***CRITICAL***: logs only critical information.
- ***DEBUG***: logs all the information including critical, error and warning.
- ***ERROR***: logs only error message.
- ***INFO***: logs basic execution information.
- ***WARNING***: logs only warning messages.

mergerSettings

In mergerSettings, configure feed data processing parameters including batch size, collection time, and the number of parallel data channels to optimize data aggregation efficiency.

`inputBufferSize`: set the number of records to read the feed data from various channels for merge process.

`maxBlockingTimeInMillis`: configure the maximum duration for gathering feed data before initiating the merge process.

`Parallel Channels`: configure the number of concurrent data streams to process feed data. In Kubernetes, this can dynamically scale up to 10 channels, while in virtual machines, a fixed number of channels can be initiated simultaneously.

`responseMessageKafkaSourceSettings`: set the number of messages processed in each batch as Tomcat consumes data from Kafka topics and sends acknowledgment messages to the Campaign Actuator.

triggerEvaluatorSettings

In the triggerEvaluatorSettings section, you can define the time lag to process data and trigger time for scheduler interval .

`batchSize`: define the batch size of the events to be executed based on the configuration in the Contact policy. Preferred batch size for production environment is "100", and for test environment, it should be "one".

`maxAllowedTimeLagInSeconds`: set the maximum acceptable age for data to be processed for the trigger evaluation. If data is older than this threshold, the trigger is halted. For instance, a value of 86400 seconds, which is 1 day old, would prevent processing data older than one day data.

`stateExpirationCheckIntervalInMillis`: configure the frequency of checking reminder-based trigger intervals. This setting controls how often the system evaluates if a reminder-based trigger should be activated.

`useKafkaToProcessActuation`: by default, this is set as true for production. In case of development activity or test, this field can be set as false.

Feed Applications

This section allows configuring feed applications and their corresponding parameters to establish data input sources.

```
"feedApplications": [
            {
                "logLevel": "INFO",
                "name": "Ericsson Usage"
            },
            {
                "logLevel": "INFO",
                "name": "Topup Demo"
            },
            {
                "kafkaSinkBatchSize": 1,
                "kafkaSourceSettings": {
                    "batchSize": 1,
                    "epochSize": 1,
                    "topicName": "RECHARGE"
                },
                "logLevel": "DEBUG",
                "name": "Recharge",
                "numParallelChannels": 1,
                "parameters": [
                    {
                        "name": "bootstrapServers",
                        "stringListValue": {
                            "values": [
                                "127.0.0.1:9092"
                            ]
                        }
                    },
                    {
                        "name": "groupId",
                        "stringValue": "group-hcl"
                    },
                    {
                        "name": "schemaRegistryUrl",
                        "stringValue": ""
                    },
                    {
                        "name": "saslKerberosKeytab",
                        "stringValue": ""
                    },
                    {
                        "name": "saslKerberosPrincipal",
                        "stringValue": ""
                    },
                    {
                        "name": "saslKerberosServiceName",
                        "stringValue": ""
                    },
                    {
                        "name": "saslMechanism",
                        "stringValue": ""
                    },
                    {
                        "name": "schemaFile",
                        "stringValue": ""
                    },
                    {
                        "name": "securityProtocol",
```

```
                        "stringValue": ""
                    }
                ]
            },
            {
                "name": "Customer Profile Refresh",
                "numParallelChannels": 1,
                "parameters": [
                    {
                        "name": "inputFileDelimiter",
                        "stringValue": "~"
                    },
                    {
                        "int32Value": 33,
                        "name": "numOfFields"
                    },
                    {
                        "name": "periodInSeconds",
                        "stringValue": "2"
                    },
                    {
                        "boolValue": true,
                        "name": "skipCampaignActuator"
                    }
                ],
                "postAggregationEnricherBatchSize": 500,
                "preAggregationEnricherBatchSize": 500
            }
        ]
```

`logLevel`: configure the level of log information required from the feed application. You can set any one of the following options:

- ***CRITICAL***: logs only critical information.
- ***DEBUG***: logs all the information including critical, error and warning.
- ***ERROR***: logs only error message.
- ***INFO***: logs basic execution information.
- ***WARNING***: logs only warning messages.

`name`: configure the feed application name. The name should match with the solution file name that is deployed in the code base.

`parameters`: define parameter name and value to be passed to the feed application. The value type can be stringValue, stringListValue, boolValue, and int32Value.

Kafka Settings

For Kafka, there are few additional fields to be configured.

`kafkaSinkBatchSize`: Specify the maximum number of processed records that feed applications will push to internal Kafka topics in a single batch.

`batchSize`: define the maximum number of records to be read at a time in single batch.

`epochSize`: configure the maximum number of records to be committed in Kafka topics in single batch after reading them.

`topicName`: set the topic name that is configured in the source Kafka topic.

`numParallelChannels`: set the number of parallel processing channels to process Kafka topics.

`"postAggregationEnricherBatchSize", "preAggregationEnricherBatchSize"`: Configure the batch size for non-real-time feed applications that process data in batches rather than a continuous stream. This setting determines the number of records bundled together before sending to Kafka topics.

Tomcat Configuration

Configure tomcat host name, port number, startup timeout, and signed certificate details.

```
"tomcatConfiguration": {
           "hostname": "{hostname}",
           "httpsEnabled": true,
           "port": 8443,
           "shutdownPort": 8009,
           "startupTimeoutInSeconds": 120,
           "usingSelfSignedCertificate": true
       }
```

`port`: set the default tomcat port number.

`shutdownPort`: set the tomcat shutdown port number.

`startupTimeoutInSeconds`: specify the maximum allowable time in seconds for Tomcat to startup.

`usingSelfSignedCertificate`: set true to use the self signed certificate for Tomcat.

Chapter 3. Configurations

Profile

An entity corresponds to a uniquely identifiable attribute in the data model. E.g. Mobile Number for a Telecom Subscriber, Cell ID for a Cell Tower, customer Id for a banking customer etc. A collection of features related to the entity. Such collection may include Static features (e.g. DOB) + Historic Features (e.g. Revenue last month) + Semi Historic Features (e.g. Dropped calls let hour) + Last-Known Features (e.g. last known location, last topup amount, etc.). The Real-time Features come in the form of Real-time events.

Profiles are technically a collection of tables in Redis, each table has a key and value is a hash map. This hash map further has key value pair, representing the profile attribute name (like gender) as key and attribute value (like Female) as value.

Profile Configuration

Profile configurations are defined in `etc/model/profiles` directory. below is the example of a solution containing two profiles definition:

```
$ tree etc/model/profiles/
etc/model/profiles/
├── customer
│   └── profile.json
└── tower
    └── profile.json
```

The name of profiles are *Customer* and *Tower*, the profile *Customer* represents telcom sunscribers and profile *Tower* represents cell tower installed by telecom providers.

The `profile.json` file is structures as below:

```
{
    keyAttribute": "MSISDN",
    "masterTable": "CUSTOMER_PROFILE",
    "name": "Customer",
    "tables": [
        {
            "name": "CUSTOMER_PROFILE",
            "type": "Dynamic"
        },
        {
            "name": "CUSTOMER_SEGMENT",
            "type": "QuasiStatic"
        }
    ]
}
```

The above definition of `Customer` profile is configured that the `keyAttribute` is *MSISDN* (Mobile number), This profile contains two tables i.e., *CUSTOMER_PROFILE* and *CUSTOMER_SEGMENT*. Among them the master profile is *CUSTOMER_PROFILE*. The profile is of two types. *Dynamic* types is table that can be changed any time in real-time and attribute values are very dynamic in nature like *lastTransactionAmount*. whereas a *QuasiStatic* table could changes slowly

like *customerSegment*. The clients which accesses the profile will invalidate the cache based on type of the profile table. a Dynamic table will be accessed each time by discarding cache, whereas a QuasiStatic tables will keep the cache for 10 minutes.

Once the profile are defined in *etc/model/profiles* directory, we need to change the product's configuration to point to these profiles. The profiles are configured in `drive.profiles` section of product configuration `etc/drive.json`:

```
..
..
"drive": {
    "profiles": [
          "Customer",
          "Tower"
      ],
}
..
..
"masterProfile": "Customer",
..
..
```

Note:

- Directory structure and naming of profile directory is very important as Detect will use this to navigate and access a particular profile configuration. The profile `Customer` is kept under *profiles/customer/profile.json* , if the profile name was `Event Catelog` then the directory structure will be *profiles/event_catelog/profile.json*.
- We can have as many profiles as required by a solution but there will one profile which is called master profile. This master profile is used as a central entity profile and used as default profile for audience and trigger evaluation.

Reference Datasets

Reference datasets are configured to populate the profile tables. This defines the data model and format of reference data along with the destination profile and profile table. The reference data generally comes from the data warehouses, operational systems or given as a static files.

below is the example of a solution containing one reference datasets definition:

```
$ tree etc/model/reference_datasets/
etc/model/reference_datasets/
├── towers.json
└── towers.schema.json
```

The name of reference dataset is *Towers*, this reference dataset is used to periodically update profile *Tower*.

The `towers.schema.json` file is structures as below:

```
{
    "attributes": [
        {
```

```
            "name": "cellId",
            "type": "String"
        },
        {
            "name": "city",
            "skipped": true,
            "type": "String"
        },
        {
            "name": "cityCode",
            "type": "String"
        },
        {
            "name": "date",
            "type": "String",
            "allowedToBeNull": true
        },
        {
            "name": "district",
            "type": "String"
        },
        {
            "name": "equipment",
            "type": "List(String)"
        },
        {
            "name": "lat",
            "type": "Double"
        },
        {
            "name": "lon",
            "type": "Double"
        },
        {
            "name": "siteType",
            "type": "String"
        },
        {
            "name": "towerType",
            "type": "String"
        }
    ],
    "dataFileFormat": "JSON",
    "dropTableBeforeInsertion": true,
    "keyAttribute": "cellId",
    "name": "Towers",
    "profile": "Tower",
    "refreshIntervalInMillis": 6000000,
    "refreshable": true,
    "table": "TOWER"
}
```

- `attributes` section defines the name, type, null constrains, skipped flag. by default all attributes are non-nullable. If an attribute is marked as *"skipped": true* then that attribute will be skipped while loading the profile table. The order of attributes in the `attribute` section defines the order of the attribute in the reference dataset's data file.
- `dataFileFormat` are of two types i.e., *JSON* or *CSV*.

- `dropTableBeforeInsertion` is flag to drop the table before loading a new file.
- `keyAttribute` is one of the attribute from the file to be used as key while loading data into profile table.
- `name` is the name of reference data set.
- `profile` is the name of profile to be loaded.
- `refreshIntervalInMillis` is the frequency of checking if the file is modified.
- `refreshable` is a flag to enable the refreshability of the reference dataset.
- `table` is the name of table among the tables list from the configured profile.

The structure of CSV file will have some additional parameters as below:

```
{   ..
        ..
        "dataFileFormat": "CSV",
    "dropTableBeforeInsertion": false,
    "keyAttribute": "cardType",
    "name": "Card Logo",
    "parameters": [
        {
            "name": "delimiter",
            "type": "String",
            "value": "|"
        },
        {
            "name": "hasHeader",
            "type": "Bool",
            "value": true
        }
    ],
        ..
        ..
}
```

- `delimiter` is the delimiter in the CSV file.
- `hasHeader` is flag to know if the file has header or not.

Once the reference datasets are defined in *etc/model/reference_datasets* directory, we need to change the product's configuration to point to these reference datasets. The reference datasets are configured in `drive.referenceDatasets` section of product configuration `etc/drive.json`:

```
..
..
"drive": {
    "profiles": [
          "Customer",
          "Tower"
      ],
      "referenceDatasets": [
          "Towers"
      ]
}
..
..
"masterProfile": "Customer",
```

```
..
..
```

Note:

- File naming of reference datasets directory is very important as Detect will use this to navigate and access a particular reference dataset configuration. The reference dataset `Towers` is kept under *reference_datasets/towers.schema.json* and since it is a *JSON* type, the Detect will expect `towers.json` in the same directory. If the type was *CSV*, then Detect will expect `towers.dat` file.
- Detect maintains a hash of file in the database in order to track changes in the data file and will only refresh if the file is modified.

Feed Data Model

Feed data models represents common classes of feed data sources in the system. E.g., *Card Transaction* feed data model will have common attributes from both *Debit Card Transactions* and *Credit Card Transactions* real-time data feed. A feed application can follow one or more of these feed data models. like *Credit Card Transactions* data feed could follow *Card Transaction* and *Location* feed data model.

feed data models are configured in `etc/models/feed_data_models/<model-name>/data_model.json`.

below is the example of a solution containing four feed data model definition:

```
$tree etc/model/feed_data_models/
etc/model/feed_data_models/
├── identity
│   └── data_model.json
├── top_channels_prediction
│   └── data_model.json
├── topup
│   └── data_model.json
└── usage
    ├── data_model.json
    ├── enrichment_functions.json
    └── enrichment_scorers.json
```

The name of feedDataModels are *Identity*, *Top Channel Prediction*, *Topup* and *Usage*.

Example Feed data model:

```
{
    "attributes": [
        {
            "name": "MSISDN",
            "required": true,
            "type": "String"
        },
        {
            "name": "calledNumber",
```

```
            "required": false,
            "type": "String"
        },
        {
            "name": "callingNumber",
            "required": false,
            "type": "String"
        },
        {
            "name": "lastMSISDN",
            "required": false,
            "type": "String"
        },
        {
            "name": "ts",
            "required": true,
            "type": "Int64"
        }
    ],
    "name": "Identity",
    "profiles": [
        {
            "keys": [
                "MSISDN",
                "calledNumber",
                "callingNumber",
                "lastMSISDN"
            ],
            "name": "Customer"
        }
    ]
}
```

- `attributes` is a list of attributes of the model.
- `name` is the name of feed data model.
- `profiles` is a list of profile that can be realted to this feed model.
- `profiles.name` is the name of profile which can be related to this feed model for lookup or update purpose.
- `profiles.keys` is the list of attribute which can be used as keys to perform operation for the given profile. one of these keys should be not nullable.

Once feed data models are configured in feed_data_models directory, we need to point to it in product configuration (`etc/drive.json`)

The feed data models are configured in `drive.feedDataModels` section of product configuration `etc/drive.json`:

```
..
..
"drive": {
    "feedDataModels": [
          "Identity",
          "Top Channels Prediction",
          "Topup",
          "Usage"
      ],
}
```

```
..
..
"masterProfile": "Customer",
..
..
```

Note:

- Directory structure and naming of feed data model directory is very important as Detect will use this to navigate and access a particular feed data model's configuration. The feed data model `Identity` is kept under *feed_data_models/identity/data_model.json* , if the feedDataModel name was `Top Channels Prediction`then the directory structure will be *feed_data_models/top_channels_prediction/data_model.json*.

Feed Applications Models

Each feed application have to configure application model and can optionally configure some other feed application model file as described below.

feed application models are configured in `etc/models/applications/<feed-name>/`.

below is the example of a solution containing a feed application definition:

```
$ tree etc/model/applications/ericsson_usage/
etc/model/applications/ericsson_usage/
├── aggregations.json
├── application.json
├── enrichment_functions.json
└── enrichments.json
```

The name of feed application is *Ericsson Usage* .

Below sub-sections will explain each of the above files.

Application Model

Application model defines the key attribute and feed data model that this feed implements, and it is defined in *application.json* file.

Example application Model:

```
{
    "feedApplication": {
        "dataModels": [
            "Identity",
            "Usage"
        ],
        "keyAttribute": "MSISDN",
        "name": "Ericsson Usage"
    }
}
```

- `dataModels` a list of feed data model that are implemented by this feed.
- `keyAttribute` an attribute from the feed that is a key attribute.
- `name` is name of feed application.

Aggregations

This model describes the aggregations that are computed on a feed and stored in FastPast. Aggregates are configured in `etc/model/applications/<application_name>/aggregation.json` file.

Example Aggregation definition is as below:

```
{
    "aggregations": [
        {
            "name": "avgCallDuration",
            "operation": {
                "aggregationKind": "Average",
                "groupByAttributes": [
                    "callingNumber"
                ],
                "valueAttribute": "duration"
            }
        },
        {
            "name": "numCallsByCell",
            "operation": {
                "aggregationKind": "Sum",
                "groupByAttributes": [
                    "cellId"
                ],
                "value": 1.0
            }
        },
        ..
        ..
    ]
}
```

- `aggregations` List of aggregations.
- `name` A name (aggregations defined on different feeds can have the same name) that is in the form of an identifier (aNameLikeThis).
- `groupByAttributes` A potentially empty list of group by attributes.
- `aggregationKind` An aggregation kind (Average, Maximum, Minimum, Sum, Variance).
- `valueAttribute` or `value` A value (such as "1" for counting) or value attribute (such as "callDuration").
- `tupleFilter` An optional tupleFilter, which is a Boolean UEL expression

Loaded once during Detect initialization, managed from the UI there on.

Enirchment Functions

Enrichment Function model file defines the meta data of an enrichment function. It defines the signature of a python function to be used in an enricher.

Enrichment Function definition is configured in `etc/model/applications/<application_name>/enrichment_functions.json` file.

Example Enrichment Function:

```
{
    "functions": [
        {
            "module": "acme.application_helpers.ericsson_usage.enrichment_helpers",
            "name": "device_change",
            "parameters": [
                {
                    "name": "deviceName",
                    "required": false,
                    "type": "String"
                },
                {
                    "name": "isSmartphone",
                    "required": false,
                    "type": "Bool"
                }
            ],
            "type": "Bool",
            "usedAttributes": [
                {
                    "name": "IMEI",
                    "required": true,
                    "type": "String"
                },
                {
                    "name": "lastIMEI",
                    "required": true,
                    "type": "String"
                }
            ]
        },
        ..
        ..
    ]
}
```

signature of python function is as below:

```
def device_change_(tuple_, deviceName=None, isSmartphone=None):
    """Returns the last time the subscriber has changed their device"""
    imei = tuple_.IMEI
    lastIMEI = tuple_.lastIMEI
    ..
    ..
    return xyz
```

- `functions` list of functions defined.
- `module` a python module path.
- `name` name of enrichment function.
- `type` return type of the function.

- `parameters` list of additional parameters to enrichment function, tuple is a default and first parameter to function. we need to only mention any additional parameters/
- `usedAttributes` the list of attributes used from tuple. This is required for Detect to know if a particular function can be applied on a feed or not.

Enrichments

This model describes the enrichments that are performed on a feed, whose results may be stored in PinPoint. Enrichments are configured in `etc/model/applications/<application_name>/enrichments.json`. It has list of enrichments, where each enrichment has: * A name (enrichments defined on different feeds can have the same name)

- A list of transformed attributes
- A list of lookup attributes
- A list of aggregated attributes
- A list of derived attributes

There are two kinds of enrichments: Pre-aggregation are list of enrichments that are applied before aggregation takes place. Post-aggregation are list of enrichments that are applied after aggregation takes place.

This file is loaded once during Drive initialization, managed from the UI there on.

Enrichment structure is as below:

```
{
    "preAggregationEnrichments": [
        {
            "name": "IMEIChangeEnrichmentPre",
            "operation": {
                "derivedAttributes": [
                    ..
                    ..
                ],
                "aggregatedAttribute": [
                    ..
                    ..
                ],
                "lookupAttribute": [
                    ..
                    ..
                ],
                "transformedAttributes": [
                    ..
                    ..
                ]
            }
        },
        {
            ..
        }
    ],
    "postAggregationEnrichments": [
        {
            "name": "IMEIChangeEnrichmentPost",
```

```
            "operation": {
                "derivedAttributes": [
                    ..
                    ..
                ],
                "aggregatedAttribute": [
                    ..
                    ..
                ],
                "lookupAttribute": [
                    ..
                    ..
                ],
                "transformedAttributes": [
                    ..
                    ..
                ]
            }
        },
        {
            ..
        }
    ]
}
```

Transformed Attributes

Its an UEL expression to derive a value for a new attribute.

Example:

```
 {
    "expression": "\"FOO\"",
    "name": "identity",
    "retained": false,
    "type": "String"
}
```

- `expression` UEL expression for to derive value.
- `name` name of enriched attribute.
- `retained` flag to retain this new attribute in the tuple for next operator in the flow.
- `type` is the data type of this attribute.

Lookup Attributes

This is used to look-up from a profile.

Example:

```
{
    "name": "licenseId",
    "retained": false,
    "tableAttribute": {
        "keyAttribute": "identity",
        "name": "licenseId",
        "table": "CUSTOMER"
```

```
    },
    "type": "String"
}
```

- `tableAttribute` is to define the table and attribute to look-up.
- `tableAttribute.keyAttribute` is key attribute in the tuple which will be used to perform lookup from table.
- `tableAttribute.name` is the name attribute from table to be looked up.
- `tableAttribute.table` is the table name form pinpoint to be looked up.
- `type` is the data type of the enrichmented attribute.

Aggregated Attributes

This enrichment attribute gets value by fetching a aggregate from FastPast.

Example:

```
{
    "aggregate": "sumOfTransactionAmountByCustomerAndTransactionType",
    "groupByAttributes": [
        "customerId",
        "transactionType"
    ],
    "name": "transactionAmountInLast7Days",
    "period": "Last",
    "retained": true,
    "type": "Double",
    "windowLength": 7,
    "windowLengthUnit": "Day"
}
```

- `aggregate` is the name of aggregate in the FastPast.
- `groupByAttributes` is a list of attributes to be used for passing for groupby attributes.
- `name` is the name of enriched attribute.
- `period` is the period for FastPast query. e.g. *Current*, *Last* or *AllTime*.
- `windowLength` is length of window for a given unit.
- `windowLengthUnit` is the unit for which query to be made. e.g., *Day*, *Month*, *Year*, *Minute*, *Hour*

Derived Attributes

The derived attributes enrichment enables the execution of an external Python function. One such a function takes in a tuple (as well as any other required additional parameters, if any), performs a user-defined computation, and produces a result. This function invocation's return value can then be retained and forwarded as part of an outgoing tuple.

Example:

```
{
    "function": {
        "module": "acme.application_helpers.ericsson_usage.enrichment_helpers",
        "name": "device_change"
    },
    "name": "deviceChanged",
    "retained": true,
```

```
    "type": "Bool"
},
{
    "function": {
        "module": "acme.application_helpers.ericsson_usage.enrichment_helpers",
        "name": "device_change_time"
    },
    "name": "deviceChangeTime",
    "retained": true,
    "storeBackAttribute": {
        "keyAttribute": "callingNumber",
        "name": "lastIMEIModificationTime",
        "table": "CUSTOMER_PROFILE"
    },
    "type": "Int64"
},
```

- `function` This refers to enrichment function model discussed in previous sub section.
- `storeBackAttribute` the derived attribute can optionally stored back to profile in pinpoint.

Events

Events are rules and associated actions configured on data streams to detect in real-time. This consist of selecting an audience of an event, configuring the certain transaction or activity to be performed by users and configuring the actions to be performed upon event detection. This section will explain how to pre-configure different templates for audience, triggers, offer actions and e2e event templates so that event creation by marketing user will be performed easily from UI.

Below are the concepts used in the event configuration process.

Enums Type Definitions

Enums are custom types, used for enabling drop-downs in audience, triggers and action offers templates. Enum types are of two types i.e., Enum and TreeEnums. Enums are configured in file `etc/model/campaigns/enum_type_definitions.json`.

Enum

This Enum Types are simple one item selection type Enums.

Lets take example of an enum and how its configuration looks like:

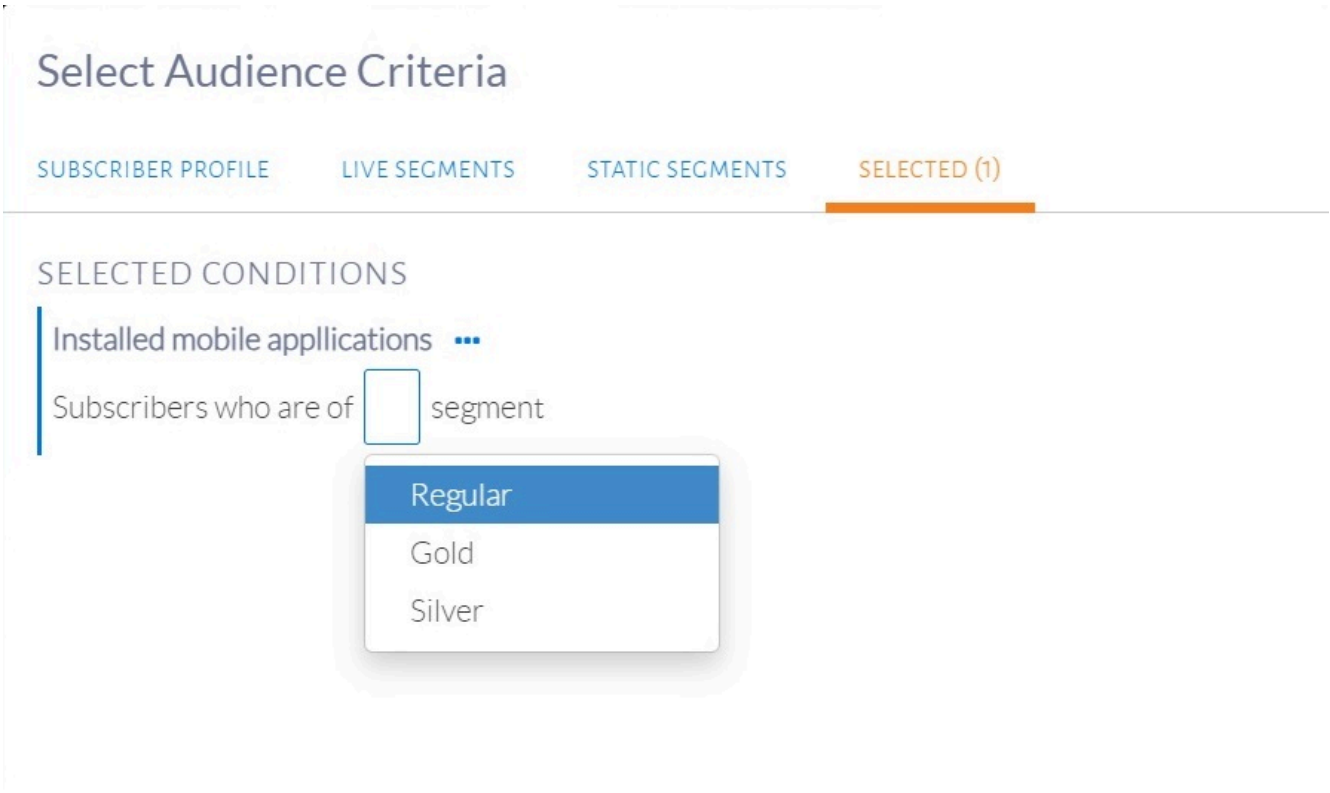
```
"enums": [
    ...
    ...
    {
          "enumName": "SubscriberCategoryType",
          "placeholder": "Select subscriber category",
          "possibleValues": [
              {
                  "displayOrder": 0,
                  "displayValue": "Regular",
                  "intEnumValue": 0
              },
              {
                  "displayOrder": 1,
```

```
                "displayValue": "Silver",
                "intEnumValue": 1
            },
            {
                "displayOrder": 2,
                "displayValue": "Gold",
                "intEnumValue": 2
            }
        ],
        "uelType": "Int32"
    },
    ...
    ...
]
}
```

Above Enum is a configuration of Subscriber Category, there are 4 possible values for this Enum i.e., Regular, Silver and Gold. In the streaming data/Real data these different values are represented as ID field and value 0 denotes *Regular*, 1 denotes *Silver* and 2 denotes *Gold*. But on UI users will descriptive values like *Gold*, *Regular*.

- `enums` is the type of enum being configured.
- `enumName` is not name of enum to be used in various condition templates.
- `placeholder` is the text to be show in place where selection is required on UI.
- `possibleValues` are list of possible values for this enum.
- `displayOrder` is order of this value in the drop-down.
- `displayValue` is value to be show in the drop-down item.
- `intEnumValue` or `stringEnumValue` or `booleanEnumValue` are the actual data value being selected.
- `uelType` is the type of values.

When we use this enum type in a template condition then UI will look like below:



Enum used in a template.

TreeEnums

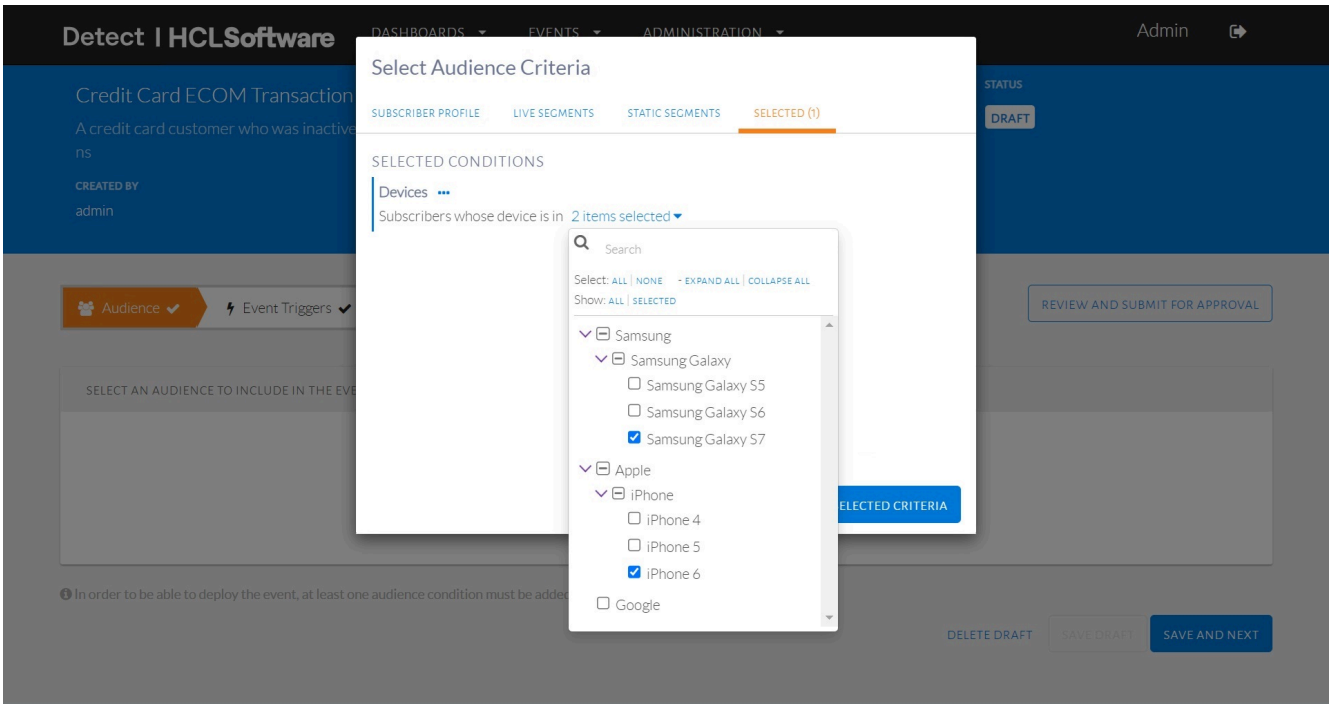
TreeEnums are hierarchical enums and allows multi-select in the drop-down. Once we select from a TreeEnum a list of values are selected. If you select a leaf node of the tree then only single value will be selected, but if you select any other node except leaf node, then all leaf node under that will be selected.

lets take example of DeviceType Enum as configured below:

```
..
..
{
    "enumName": "Devices",
    "placeholder": "Select device",
    "possibleValues": [
        {
            "children": [
                {
                    "children": [
                        {
                            "displayOrder": 0,
                            "displayValue": "Samsung Galaxy S5",
                            "enumValue": "GALAXY_S5"
                        },
                        {
```

```
                            "displayOrder": 1,
                            "displayValue": "Samsung Galaxy S6",
                            "enumValue": "GALAXY_S6"
                        },
                        {
                            "displayOrder": 2,
                            "displayValue": "Samsung Galaxy S7",
                            "enumValue": "GALAXY_S7"
                        }
                    ],
                    "displayOrder": 0,
                    "displayValue": "Samsung Galaxy",
                    "enumValue": "ALL_GALAXY_S"
                }
            ],
            "displayOrder": 0,
            "displayValue": "Samsung",
            "enumValue": "ALL_SAMSUNG"
        },
        {
            "children": [
                {
                    "children": [
                        {
                            "displayOrder": 0,
                            "displayValue": "iPhone 4",
                            "enumValue": "IPHONE4"
                        },
                        {
                            "displayOrder": 1,
                            "displayValue": "iPhone 5",
                            "enumValue": "IPHONE5"
                        },
                        {
                            "displayOrder": 2,
                            "displayValue": "iPhone 6",
                            "enumValue": "IPHONE6"
                        }
                    ],
                    "displayOrder": 0,
                    "displayValue": "iPhone",
                    "enumValue": "ALL_APPLE_IPHONE"
                }
            ],
            "displayOrder": 1,
            "displayValue": "Apple",
            "enumValue": "ALL_APPLE"
        },
        {
            "displayOrder": 2,
            "displayValue": "Google",
            "enumValue": "ALL_GOOGLE"
        }
    ]
}
..
..
```

When we use this enum type in a template condition then UI will look like below:



TreeEnum used in a template.

Note: Enum type definitions are loaded in-memory every-time we restart the tomcat backend.

Audience Condition Templates

Audience Condition Templates are template condition for filtering the master profile of Detect. This selects a list of users which satisfies an audience condition.

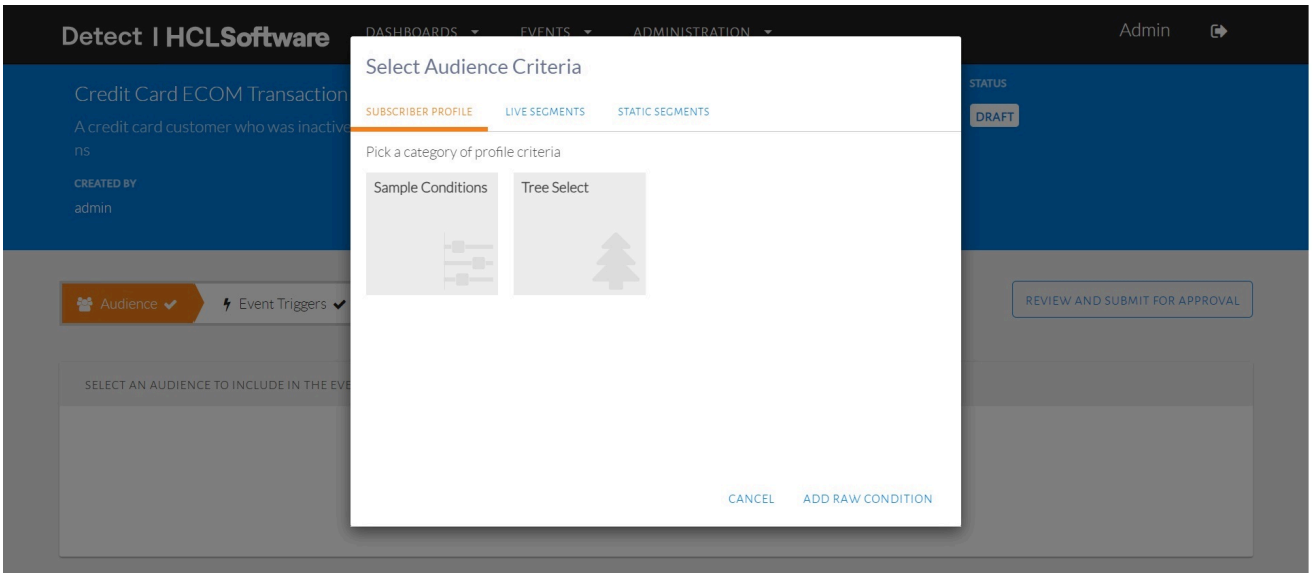
audience conditions are configured in `etc/model/campaigns/audience_condition_categories.json` file.

Audience templates are grouped into named categories based on their business logic as shown below:

```
{
    "categories": [
        {
            "icon": "fa-sliders",
            "name": "Demographic",
            "templates": [
                {
                    ...
                },
                {
                    ...
                }
            ]
        },
        {
```

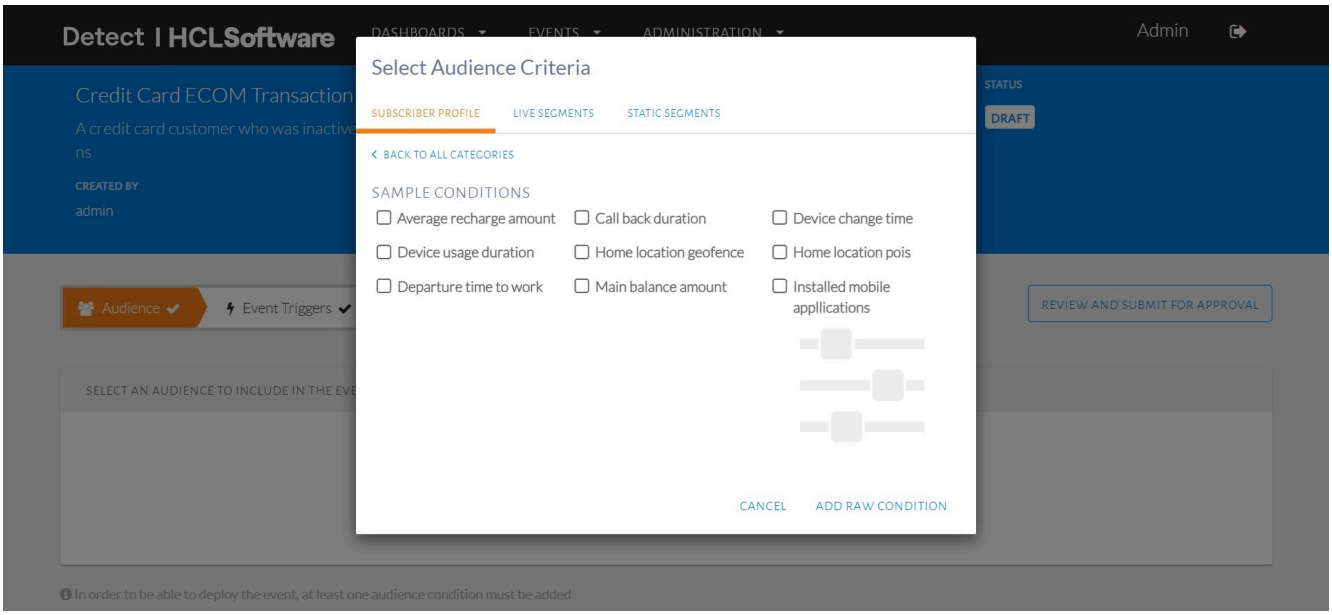
```
            "icon": "fa-user",
            "name": "Transaction Based",
            "templates": [
                {
                    ...
                },
                {
                    ...
                }
            ]
        }
}
```

categories are shown below:



Audience categories.

Once we select an audience category, all audience templates under that audience category will shown as below:



List of Audience templates.

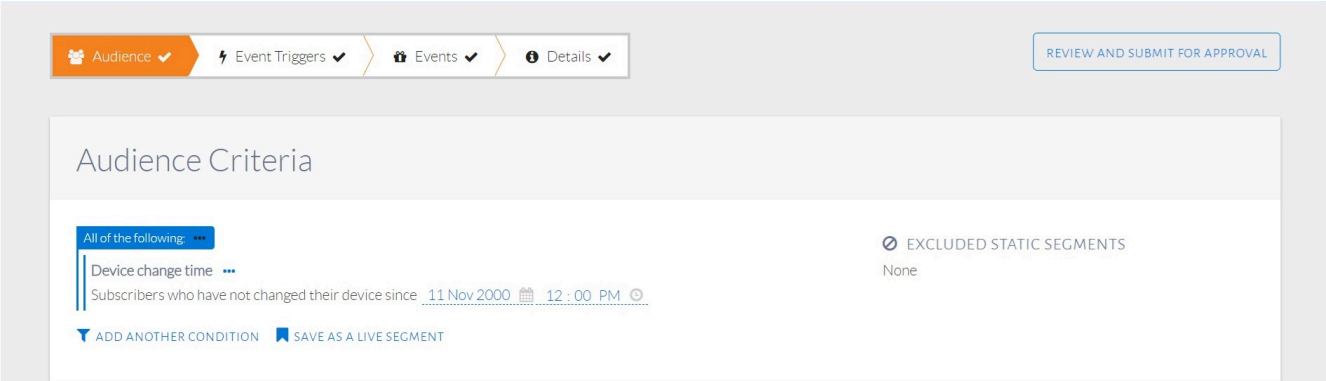
Elements inside the `templates` are different audience template condition, we will try learn how to configure audience condition templates by examples.

Simple Audience Templates

Below is an example audience template, which filters users who have not changes device since some configurable date time:

```
{
    "analyticsBased": false,
    "caption": "Subscribers who have not changed their device since ${deviceChangeDateTime}",
    "conditionTemplateId": "deviceChangeDateTime",
    "domainMapping": {
        "expressionTemplate": "profile.deviceChangeTime < ${deviceChangeDateTime}"
    },
    "name": "Device change time",
    "profile": "Customer",
    "segments": [
        {
            "segmentId": "deviceChangeDateTime",
            "segmentKind": "Complex",
            "segmentType": "DateTime",
            "segmentValue": {
                "dateTimeValue": "2000-11-11T12:00:00"
            }
        }
    ]
},
```

Lets look how it looks in the UI if we select this audience template in the use case:



Simple Audience condition.

- `caption` is the caption text which will appear in the UI. It can optionally contain one or more segmentIds in order to make caption customizable. In this example *deviceChangeDateTime* is a segmentId and definition of this segmentId is done in the `segments` section. Segments are used to get inputs from users while they are configuring an audience condition.
- `conditionTemplateId` is an unique id given to used by Detect.
- `domainMapping.expressionTemplate` is the UEL expression that always evaluates to either *true* or *false*. A Subscriber will be part of this audience only if this UEL expression evaluates to *true* based on their profile attributes. In order to access a profile attribute, `profile.` is suffixed in the name of attribute. In this example `profile.deviceChangeTime` will access *deviceChangeTime* attribute of master profile of any given subscriber.
- `name` is the name of the audience condition under a given audience category.
- `profile` should be master profile name.
- `segments` the list of segment definitions used in the caption.
- `segmentId` is id of segment used in the caption.
- `segmentKind` is the data type of segment. This can be `Primitive` type like *String*, *Integer*, *Double* or *Boolean*. or `Complex` type like Time, POIs, Geofences, Duration, DateTime, AggregationDuration, MonetaryValue. or Enum or TreeEnum.
- `segmentValue` is the value which will be shown as default value in the UI which user can update while configuring the audience.

Audience Template With Modifiers

An audience template can have one or more optional additional condition along with primary *domainMapping* condition expression. These are called `modifiers`.

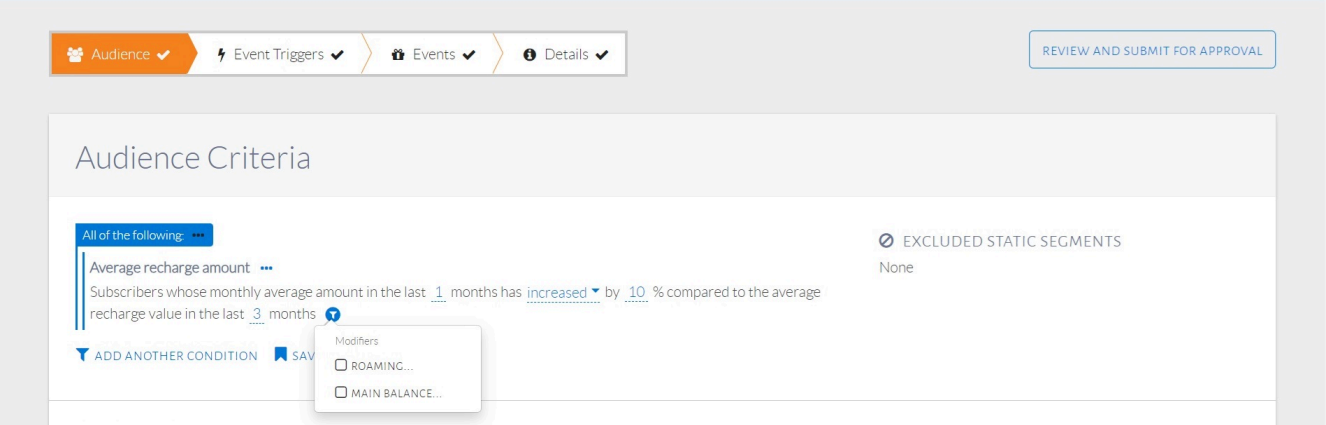
Let look at below example:

```
{
    "analyticsBased": false,
    "caption": "Subscribers whose monthly average amount in the last ${recentObservationWindowLength} months
 has ${increasedDecreasedIndicator} by ${changePercentage}% compared to the average recharge value in the last
 ${historicalObservationWindowLength} months",
    "conditionTemplateId": "averageRechargeAmount",
    "domainMapping": {
        ...
        ...
```

```
        }
    },
    "modifiers": [
        {
            "caption": "${hasRoaming} roaming enabled",
            "domainMapping": {
                "expressionTemplate": "profile.roamingEnabled == ${hasRoaming}"
            },
            "modifierId": "roamingEnabled",
            "name": "Roaming...",
            "segments": [
                {
                    "captions": [
                        "Does not have",
                        "Has"
                    ],
                    "segmentId": "hasRoaming",
                    "segmentKind": "Enum",
                    "segmentType": "Boolean",
                    "segmentValue": {
                        "booleanValue": false
                    }
                }
            ]
        },
        {
            "caption": "Main balance is greater than ${mainBalanceAmount}",
            "domainMapping": {
                "expressionTemplate": "profile.mainBalanceAmount >= ${mainBalanceAmount}"
            },
            ...
            ...
        }
    ],
    "name": "Average recharge amount",
    "profile": "Customer",
    "segments": [
        ..
    ]
}
```

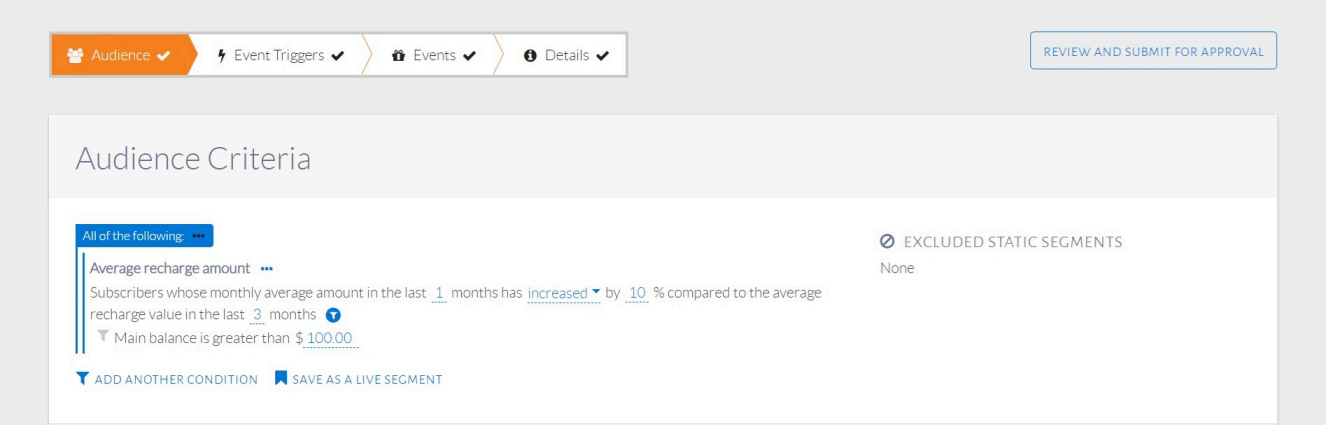
In above example we have 2 additional modifiers condition along with primary *domainMapping* condition. If selected a modifier in the UI, modifier's *domainMapping* is added as an *and* condition logic to primary *domainMapping* condition.

Lets look how the list of modifiers looks in the UI :



Modifiers list in the Audience condition.

If we select a modifier, the UI reflect it by expanding the caption and allowing users to exit default segment values.



Modifiers selected in the Audience condition.

Modifier follows the same structure as explained in simple audience template, expect it has unique `modifierId` instead offer `conditionTemplateId`

- *hasRoaming* is a segment of type *Enum*. Its Enum type is *Boolean* and selected segment value is *false*

below is an example of a condition using `TreeEnum`:

```
{
    "analyticsBased": false,
    "caption": "Subscribers who have installed appllications from categories ${appCategories}",
    "conditionTemplateId": "treeSelectAppCategories",
    "domainMapping": {
        "expressionTemplate": "profile.installedAppCategories in ${appCategories}"
    },
    "name": "App categories",
    "profile": "Customer",
    "segments": [
        {
            "segmentId": "appCategories",
```

```
            "segmentKind": "TreeEnum",
            "segmentType": "AppCategories",
            "segmentValue": {
                "stringListValue": {
                    "values": [
                        "BOOKS_AND_REFERENCE",
                        "GAME_ACTION"
                    ]
                }
            }
        }
    ]
}
```

`TreeEnum` selection returns are list, that is why the *segmentValue* is *stringListValue* type.

Audience template With Segment Switch

If we want to have different expressions to be used in a *domainMapping* based on the value of user selected *segmentId*, then we could use `segmentSwitch` in the *domainMapping*

example as below:

```
{
    "analyticsBased": false,
    "caption": "Subscribers who ${belong} to the ${staticSegment} static segment",
    "conditionTemplateId": "staticSegment",
    "domainMapping": {
        "segmentSwitch": {
            "cases": [
                {
                    "expressionTemplate": "${staticSegment} not in profile.segments",
                    "segmentValue": "false"
                },
                {
                    "expressionTemplate": "${staticSegment} in profile.segments",
                    "segmentValue": "true"
                }
            ],
            "segmentId": "belong"
        }
    },
    "name": "Static Segment",
    "profile": "Customer",
    "segments": [
        {
            "captions": [
                "do not belong",
                "belong"
            ],
            "segmentId": "belong",
            "segmentKind": "Enum",
            "segmentType": "Boolean",
            "segmentValue": {
                "booleanValue": true
            }
        },
```

```
        {
            "segmentId": "staticSegment",
            "segmentKind": "DynamicEnum",
            "segmentType": "StaticSegment"
        }
    ]
}
```

- In above example we can see that we have a *segmentIdbelong* whose value will decide which *expressionTemplate* will be used in the domain mapping.
- We can also override the display value of a Enum by addition *captions* in the segment definition.
- DynamicEnum is a built-in enum for *StaticSegment* selection.

Audience template with Aggregate condition

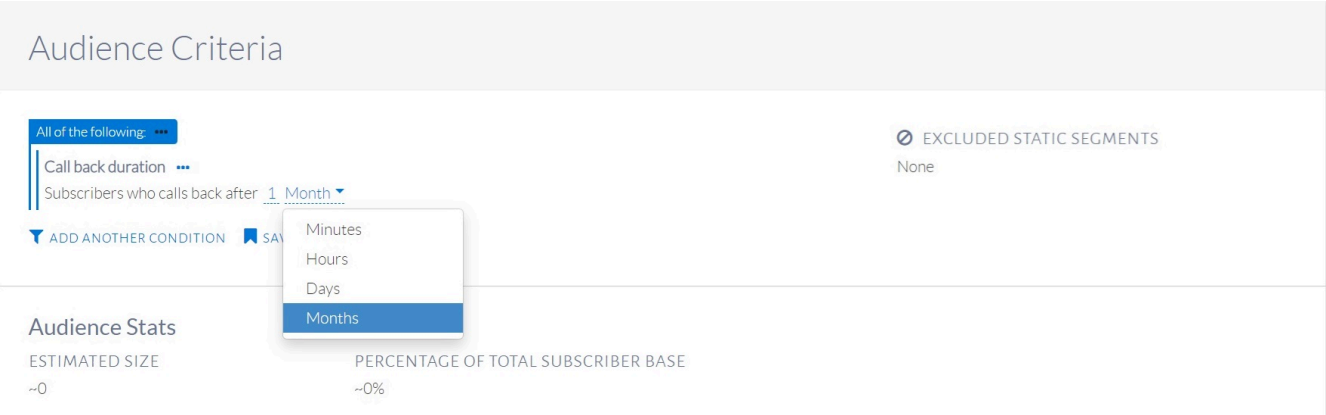
Audience condition can make use of aggregates being maintain by Detect. The condition can combine any combination of profile attribute based condition and aggregate based condition.

details of aggregates based UEL expression can be found in Miscellaneous - UEL section.

An example of aggregate based profile condition is as below:

```
{
    "analyticsBased": false,
    "caption": "Subscribers who calls back after ${callBackDuration}",
    "conditionTemplateId": "callBack",
    "domainMapping": {
        "expressionTemplate": "aggregate(numOutgoingCallsBySubscriber[profile.MSISDN], Last,
 ${callBackDuration.amount}, #{callBackDuration.windowLengthUnit}) == 5"
    },
    "name": "Call back duration",
    "profile": "Customer",
    "segments": [
        {
            "segmentId": "callBackDuration",
            "segmentKind": "Complex",
            "segmentOption": {
                "aggregationDurationOption": {
                    "durationStepSizeInMinutes": 10
                }
            },
            "segmentType": "AggregationDuration",
            "segmentValue": {
                "aggregationDurationValue": {
                    "amount": 1,
                    "unit": "Months"
                }
            }
        }
    ]
},
```

Below is the UI for the same:



Aggregate based Audience condition.

- The segment *callBackDuration* is of kind *Complex*, of type *AggregationDuration*.
- *callBackDuration.amount* will give Window Unit Length for the aggregate query.
- *callBackDuration.windowLengthUnit* will give aggregate window unit i.e., *Month*, *Year*, *Hour*, *Minute*, or *Day*.
- *aggregate(numOutgoingCallsBySubscriber[profile.MSISDN], Last, ${callBackDuration.amount}, #{callBackDuration.windowLengthUnit})*, here *numOutgoingCallsBySubscriber* is name of aggregate, *profile.MSISDN* is group by attribute, *Last* is the *period*.

Metrics

To better breakdown and understand the audience of an event, charts based on different metrics can be added. These chart show histograms on various metrics and these metrics are configured in the metric_categories.json file.

Trigger templates are grouped into named categories based on their business logic as shown below:

```
{
    "categories": [
        {
            "icon": "fa-phone",
            "name": "Voice call metrics",
            "metrics": [
                {
                    ...
                },
                {
                    ...
                }
            ]
        },
        {
            "icon": "fa-money",
            "name": "Recharge metrics",
            "metrics": [
                {
                    ...
                },
                {
                    ...
```

```
                }
            ]
        }
}
```

Elements inside the `metrics` are different metric , we will try learn how to configure metrics by examples.

metrics categories are configured in `etc/model/campaigns/metric_categories.json` file.

Profile Attributes Sourced Metric

The Metric charts prepared from Profile Attributes Sourced metrics gets data from profiles and histograms are produced from profile values of each subscriber.

Example Configuration:

```
{
    "dataType": "String",
    "metricId": "gender",
    "minimumBucketSize": 50,
    "name": "Gender",
    "profileAttributeSource": {
        "name": "gender"
    }
}
```

- `profileAttributeSource` selects a profile attribute on which histograms needs to be produced.
- `name` is name of metric.
- `dataType` is data type of X-Axis.
- `metricId` is unique name of metric.
- `minimumBucketSize` is the minimumBucketSize of histograms.

Aggregate Sourced Metrics

The Metric charts prepared from Aggregate Sourced metrics gets data from aggregate and histograms are produced from aggregate values buckets of each subscriber.

Example Configuration:

```
{
    "icon": "fa-money",
    "metrics": [
        {
            "aggregateSource": {
                "groupByAttributes": [
                    {
                        "name": "MSISDN"
                    }
                ],
                "name": "topupAmountBySubscriber"
            },
            "dataType": "Currency",
            "metricId": "topupAmount",
            "minimumBucketSize": 300,
```

```
            "name": "Topup Amount"
        }
    ],
    "name": "Recharge metrics"
},
```

- `aggregateSource` as this metric get data from an aggregate.
- `aggregateSource.groupByAttributes` the group by attribute to be passed.
- `aggregateSource.name` is the name of aggregate.

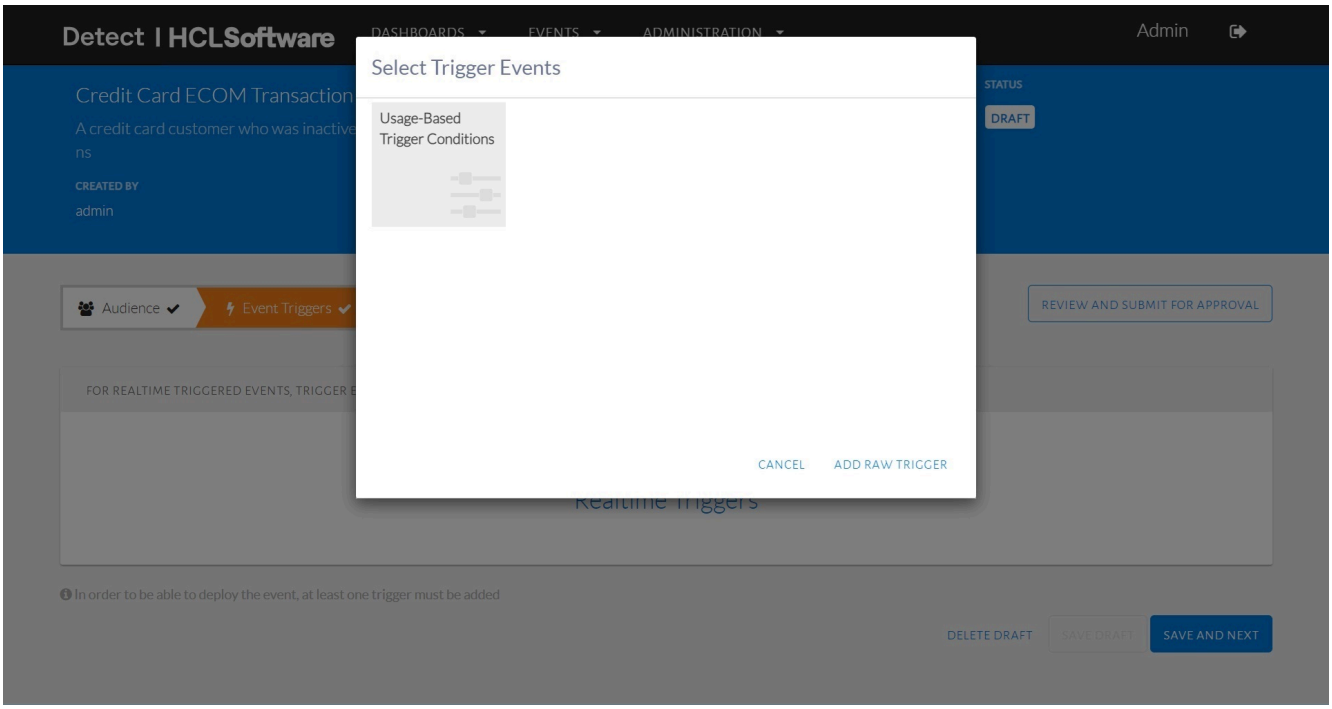
Trigger Event Categories

Event trigger profile categories is a list of categories used for defining the templates used to create trigger conditions. Each category has a name and list of templates, each template has defines a list of segments. A segment is a customizable part of the audience condition like segments in audience templates which can be used when forming the template's caption. Each trigger template also defines a domain mapping, the domain mapping may contains a switch/case statement on the segment values, and translates the condition into a UEL expression using profile attributes and aggregates, very similar to audience selection profile categories discussed earlier The former can reference profile attributes, where the latter can reference both the profile attributes and the feed attributes aka tuple's attributes.

Trigger templates are grouped into named categories based on their business logic as shown below:

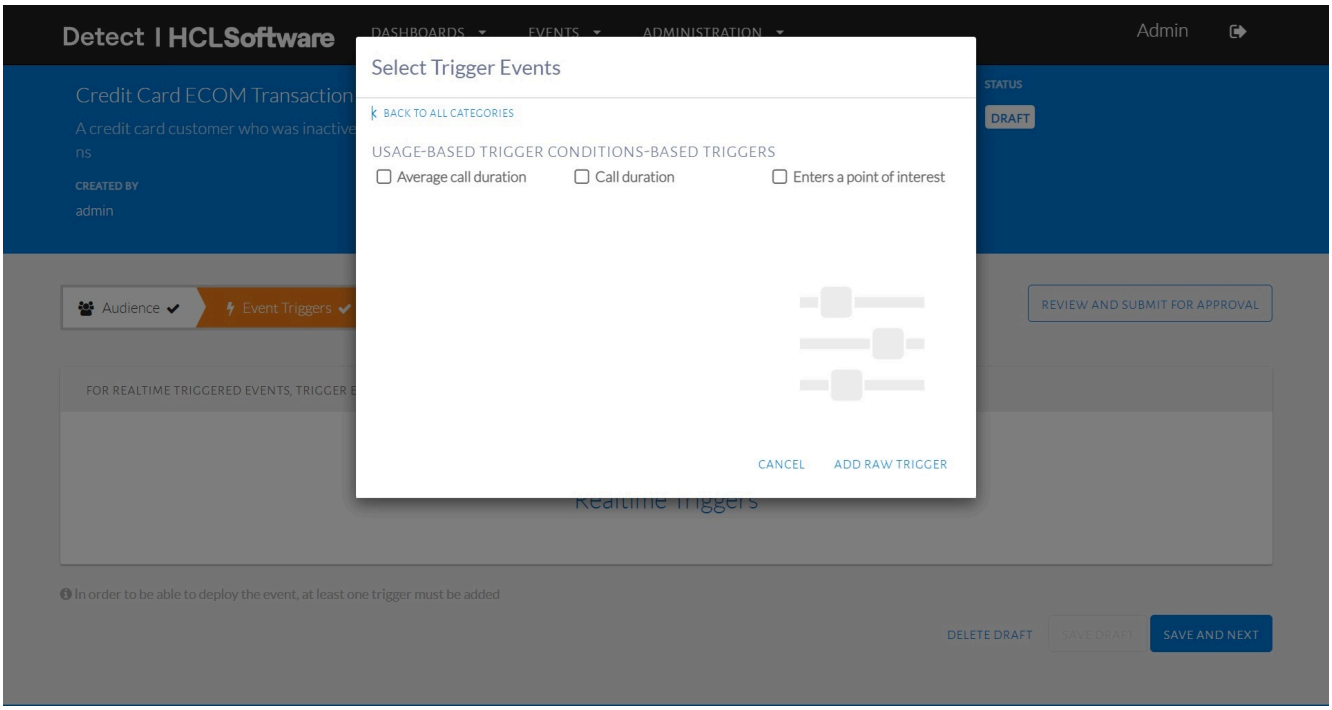
```
{
    "categories": [
        {
            "icon": "fa-user",
            "name": "Usage-Based Trigger Conditions",
            "templates": [
                {
                    ...
                },
                {
                    ...
                }
            ]
        },
        {
            "icon": "fa-crosshairs",
            "name": "Usage threshold Based",
            "templates": [
                {
                    ...
                },
                {
                    ...
                }
            ]
        }
}
```

categories are shown below:



Trigger categories.

Once we select an trigger category, all trigger templates under that trigger category will shown as below:



List of Trigger templates.

Elements inside the `templates` are different trigger template condition, we will try learn how to configure trigger condition templates by examples.

trigger conditions are configured in `etc/model/campaigns/trigger_event_categories.json` file.

As trigger templates are very similar to audience template, below section will only discuss about the difference between them.

Simple Trigger Templates

Example:

```
{
    "analyticsBased": false,
    "caption": "When the subscriber's current call duration is greater than ${callDuration}",
    "conditionTemplateId": "callDuration",
    "domainMapping": {
        "expressionTemplate": "duration > ${callDuration}"
    },
    "feedsSelector": {
        "byNames": [
            "Ericsson Usage"
        ]
    },
    "name": "Call duration",
    "segments": [
        {
            "segmentId": "callDuration",
            "segmentKind": "Primitive",
            "segmentType": "Double",
            "segmentValue": {
                "doubleValue": 1.0
            }
        }
    ]
},
```

- Here `domainMapping.expressionTemplate` uses attribute *duration*, this is an attribute in the real-time feed *Ericsson Usage*.
- `feedsSelector` select a feed on which *domainMapping* expression will be evaluated. Feeds can be selected by `byNames` i.e., a list of feed application names or by *byModelNames* i.e. a list of feed data models.

Trigger Templates With Aggregate

Example:

```
{
    "analyticsBased": false,
    "caption": "When the subscriber's current call duration is greater than ${averageCallDurationPercent}% to
 his average call duration in the last ${windowLength} hours",
    "conditionTemplateId": "averageCallDuration",
    "domainMapping": {
        "expressionTemplate": "((duration / aggregate(avgCallDuration[MSISDN], Last, ${windowLength}, Day)) *
 100 ) > ${averageCallDurationPercent}"
    },
```

```
    "feedsSelector": {
        "byNames": [
            "Ericsson Usage"
        ]
    },
    "name": "Average call duration",
    "segments": [
        {
            "segmentId": "averageCallDurationPercent",
            "segmentKind": "Primitive",
            "segmentType": "Double",
            "segmentValue": {
                "doubleValue": 2.0
            }
        },
        {
            "segmentId": "windowLength",
            "segmentKind": "Primitive",
            "segmentType": "Integer",
            "segmentValue": {
                "integerValue": 1
            }
        }
    ]
},
```

Trigger Templates With Modifiers

Example:

```
{
    "analyticsBased": false,
    "caption": "When a voice transaction using the ${walletName} wallet takes place for a subscriber",
    "conditionTemplateId": "transactionUsingAWallet",
    "domainMapping": {
        "expressionTemplate": "currentWalletName == ${walletName}"
    },
    "feedsSelector": {
        "byNames": [
            "Voice"
        ]
    },
    "modifiers": [
        {
            "caption": "and wallet is expiring within ${daysToExpire} day(s)",
            "domainMapping": {
                "expressionTemplate": "currentWalletDaysToExpire == ${daysToExpire}"
            },
            "modifierId": "daysUntilExpirationForWallet",
            "name": "Days until expiration for wallet",
            "segments": [
                {
                    "segmentId": "daysToExpire",
                    "segmentKind": "Primitive",
                    "segmentType": "Integer",
                    "segmentValue": {
                        "integerValue": 2
                    }
```

```
                }
            ]
        }
    ],
    "name": "Transaction using the wallet",
    "segments": [
        {
            "segmentId": "walletName",
            "segmentKind": "Primitive",
            "segmentType": "String",
            "segmentValue": {
                "stringValue": "Data200"
            }
        }
    ]
}
```

Offer Action Categories

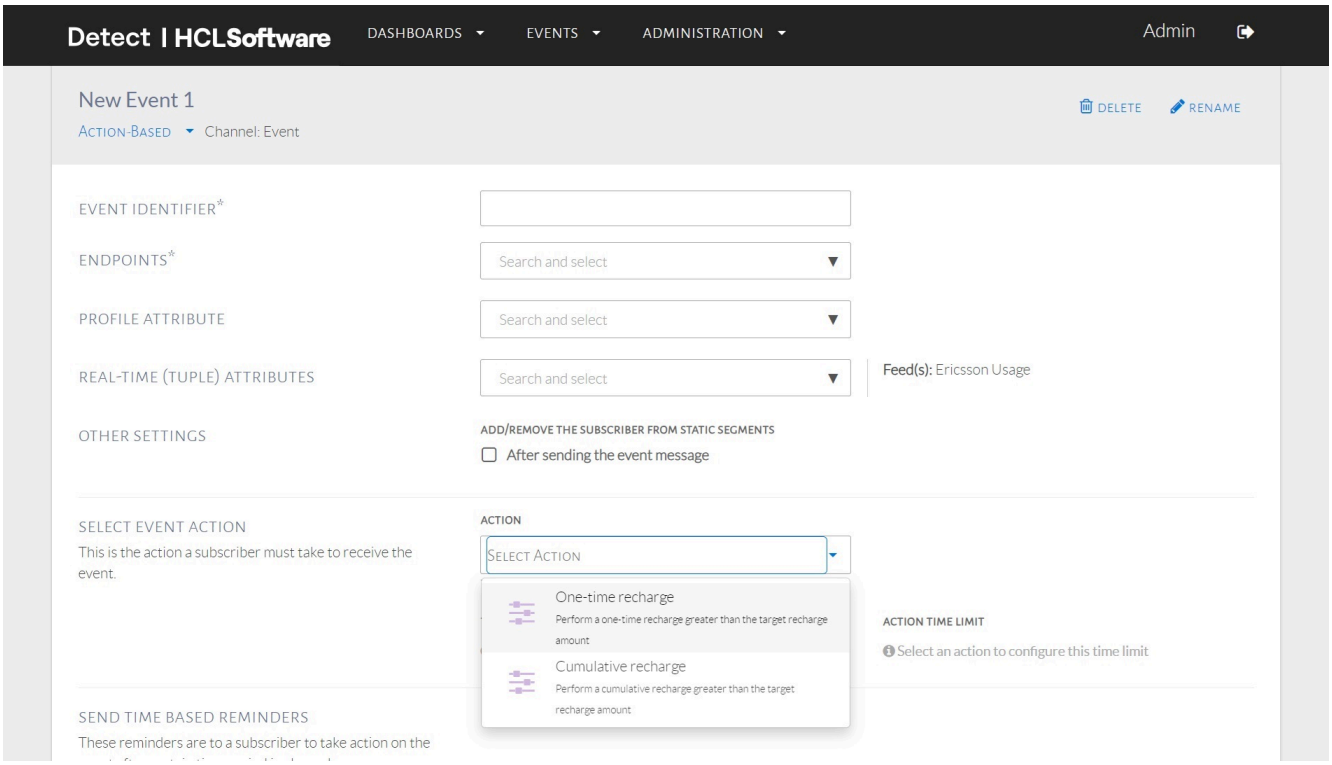
Offer action categories are configured to enable action based offers. These actions are of two types.

offer action categories are configured in `etc/model/campaigns/offer_action_categories.json` file.

Offer Action templates are grouped into named categories based on their business logics as shown below:

```
{
    "categories": [
        {
            "icon": "fa-sliders",
            "name": "Recharge-Based Actions",
            "templates": [
                {
                    ...
                },
                {
                    ...
                }
            ]
        },
}
```

Action offers are shown below:



Offer actions.

Non-Cumulative Actions

Non-Cumulative Actions are a single action to be performed by subscriber on a real time feed. Detect will track this event in real-time.

Example:

```
{
    "analyticsBased": false,
    "caption": "Perform a one-time recharge greater than the target recharge amount",
    "conditionTemplateId": "subscriberRechargeAmount",
    "feedsSelector": {
        "byNames": [
            "Topup Demo"
        ]
    },
    "name": "One-time recharge",
    "targetMetricSegment": {
        "actionType": "NonCumulative",
        "domainMapping": {
            "expressionTemplate": "sellAmount"
        },
        "name": "Recharge Amount",
        "type": "Double",
        "unit": "#{currencySymbol}",
        "value": {
            "doubleValue": 100
        }
```

```
    }
},
```

- `caption` is description text to be appeared in the drop-down list.
- `conditionTemplateId` is a unique name for a offer action template.
- `feedsSelector` select feeds where this action is expected.
- `name` is the name of offer action as it will appear in the drop-down list.
- `targetMetricSegment` contains the action definition.
- `targetMetricSegment.actionType` is type of action, this is an example for `NonCumulative` action hence only single action needed.
- `targetMetricSegment.domainMapping` is expression for the attribute or a formula, this will be compared with configured threshold value.
- `targetMetricSegment.name` is the of the attribute for UI.
- `targetMetricSegment.type` is the data type of attribute value.
- `targetMetricSegment.value` is the threshold value.

Cumulative Actions

This allows actions to be tracked over multiple transactions done by users, the values are accumulated and thresholds are compared with this cumulative value.

Example:

```
{
    "analyticsBased": false,
    "caption": "Perform a cumulative recharge greater than the target recharge amount",
    "conditionTemplateId": "cumulativeSubscriberRechargeAmount",
    "feedsSelector": {
        "byNames": [
            "Topup Demo"
        ]
    },
    "modifiers": [
        {
            "caption": "${hasRoaming} roaming enabled",
            "domainMapping": {
                "expressionTemplate": "profile.roamingEnabled == ${hasRoaming}"
            },
            "modifierId": "roamingEnabled",
            "name": "Roaming...",
            "segments": [
                {
                    "captions": [
                        "Does not have",
                        "Has"
                    ],
                    "segmentId": "hasRoaming",
                    "segmentKind": "Enum",
                    "segmentType": "Boolean",
                    "segmentValue": {
                        "booleanValue": false
                    }
```

```
                }
            ]
        }
    ],
    "name": "Cumulative recharge",
    "targetMetricSegment": {
        "actionType": "Cumulative",
        "domainMapping": {
            "expressionTemplate": "sellAmount"
        },
        "name": "Recharge Amount",
        "type": "Double",
        "unit": "#{currencySymbol}",
        "value": {
            "doubleValue": 100
        }
    }
}
```

- `targetMetricSegment.actionType` is *Cumulative*.
- We can optionally have modifier to filter transaction that should be accumulated.

Chapter 4. Integrations

Unica Interact Configuration

Unica Interact

Downstream Kafka is a distributed event store and stream-processing platform that captures events and send them to Unica Interact. You can configure the drive.json file with the kafka parameters that are necessary to map event details and ensure they are sent to the right destination. Based on the set parameters, events will display the set parameter values in the drop-down of the events files.

In the drive.json file, go to the *eventConsumers* section and set the Unica Interact parameters to send the event details. The sample configuration of downstream Kafka publishing events to Unica Interact is shown below:

```
"eventConsumers": [
  {
    "className": "com.hcl.drive.communication.UnicaInteractEventConsumer",
    "name": "UnicaInteractEventConsumer",
    "parameters": [
      {
        "name": "audienceIdAttributeName",
        "stringValue": "ACCOUNTID"
      },
      {
        "name": "campaignIdAttributeName",
        "stringValue": "offerCode"
      },
      {
        "name": "customerKeyFieldName",
        "stringValue": "MSISDN"
      },
      {
        "name": "databaseServerIp",
        "stringValue": "127.0.0.1"
      },
      {
        "name": "databaseServerUser",
        "stringValue": "drive"
      },
      {
        "name": "databaseServerPassword",
        "stringValue": "6F2FA7F6B3CD1570F19F0A0B2636E033"
      },
      {
        "int32Value": 3306,
        "name": "databaseServerPort"
      },
      {
        "name": "databaseName",
        "stringValue": "drive_acme_core"
      },
      {
        "name": "dateTimeAttributeName",
        "stringValue": "ts"
```

```
      },
      {
        "name": "eventParamEventName",
        "stringValue": "event"
      },
      {
        "name": "eventParamInboundChannelName",
        "stringValue": "inbound_channel"
      },
      {
        "name": "eventParamInboundChannelValue",
        "stringValue": "hcl"
      },
      {
        "name": "payloadChDebugValue",
        "stringValue": "false"
      },
      {
        "name": "payloadGatewayValue",
        "stringValue": "InboundGateway"
      },
      {
        "name": "payloadIcNameValue",
        "stringValue": "testDetect"
      },
      {
        "name": "producerConfigFilePath",
        "stringValue": "${HCL_HOME}/../${HCL_SOLUTION}/etc/producer_config.properties"
      },
      {
        "name": "producerKrb5ConfigFilePath",
        "stringValue": "/etc/krb5.conf"
      },
      {
        "name": "producerTopic",
        "stringValue": "INTERACT_QUALIFIED_TRANS_TOPIC"
      }
    ]
  }
]
```

1. *eventConsumers.className*: Name of the Java class to write the events in the pre-defined format of the downstream system. For Unica Interact, the className must be set as "*com.hcl.drive.communication.UnicaInteractEventConsumer*".
2. *eventConsumers.name*: Name of the downstream system to be assigned.
3. Parameters:
    a. Database related:
        i. *databaseServerIp*: Server IP of the database to publish downstream events to.
        ii. *databaseServerUser*: Username of the database.
        iii. *databaseServerPassword*: Encrypted password of the database.
        iv. *databaseServerPort*: Port number of the database.
        v. *databaseName*: Name of the database.
    b. Kafka related:

i. *producerConfigFilePath*: Path to the producer_config.properties file. This file would have the configuration/properties for connecting to Kafka.
ii. *producerKrb5ConfigFilePath*: Path to the krb5.conf file if needed to establish the Kafka connection that needs to be Kerberos authenticated.
iii. *producerTopic*: Name of the Kafka Topic on which events will be published, and Unica Interact will read from it.

c. *audienceIdAttributeName*: Set the audience name to be passed to Unica Interact.
d. *campaignIdAttributeName*: Set the event ID, if required, set multiple event IDs.
e. *customerKeyFieldName*: Pass the CED attribute name, which represents the Audience in Unica Interact.
f. *dateTimeAttributeName*: Set the attribute as "datetime" to pass the date information to Unica Interact. It is necessary for the attribute to be selected on the UI Event for pushing to downstream.

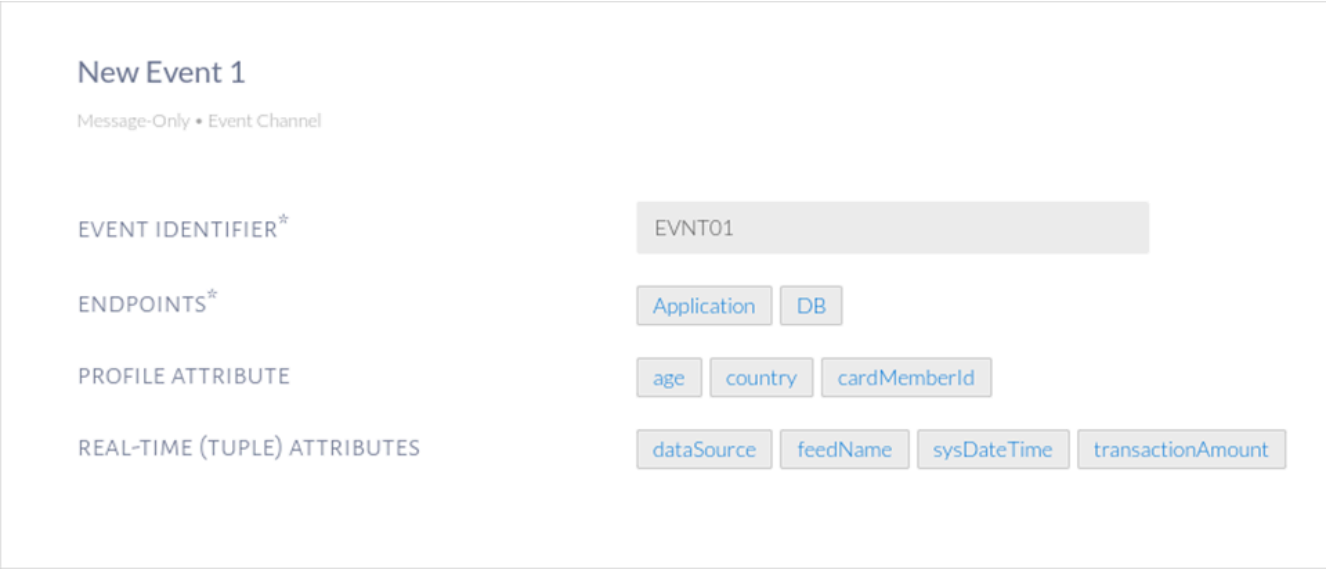
   Sample date format is shown below:

```
events: [
  parameters: [
    {
      "t": "datetime",
      "n": "ts",
      "v": "1715581447"
    }
  ]
]
```

g. *dateTimeAttributeFormat*: Attribute format of the datetime when time-based reminders are configured for an Event. By default, it is set as "yyyy-MM-dd HH:mm:ss". For custom patterns you are verify here.
h. *eventParamEventName*: Name of the parameter on which the Event ID (which is configured from UI) is set.
i. *eventParamInboundChannelName*: Attribute name of the inbound channel, which Unica Interact will use.
j. *eventParamInboundChannelValue*: Value of the attribute for inbound channel, which Unica Interact will use.
k. *payloadChDebugValue*: set the value of "CH_Debug" to be passed to Unica Interact.
l. *payloadGatewayValue*: set the value of "gateway" to be passed to Unica Interact.
m. *payloadIcNameValue*: set the value of "ICName" to be passed to Unica Interact.
n. *producerConfigFilePath*: set the config file path.

Sample Message

A sample screenshot shown below within an event configured in Detect where we select the attributes and populate an event identifier to push to downstream kafka topic for Unica Interact to ingest.



Sample message output to the Kafka topic based on the configuration above.

```
{
  "message": {
    "parameters": [],
    "ICName": "testDetect", # Value from payloadIcNameValue
    "CH_debug": "false", # Value from payloadChDebugValue
    "CH_sessionID": "8bd6a003-8ef3-4b2e-a540-3cb936a74297",# Random unique uuid
    "audienceID": [
      {
        "t": "string",
        "n": "CUSTOMER", # Value from audienceIdAttributeName
        "v": "9881153226" # Value from tuple[customerKeyFieldName]
      }
    ],
    "events": [
      {
        "parameters": [
          {
            "t": "string",
            "n": "event", # Value from eventParamEventName
            "v": "EVNT01"
          },
          {
            "t": "string",
            "n": "inbound_channel", # Value from eventParamInboundChannelName
            "v": "hcl" # Value from eventParamInboundChannelValue
          },
          { # Profile Attribute
            "t": "string",
            "n": "cardMemberId",
            "v": "12458"
          },
          { # Profile Attribute
            "t": "string",
            "n": "country",
            "v": "USA "
          },
```

```
          { # Profile Attribute
            "t": "numeric",
            "n": "age",
            "v": "22"
          },
          { # Tuple Attribute
            "t": "string",
            "n": "sysDateTime",
            "v": "2024-05-12 07:21:31"
          },
          { # Tuple Attribute
            "t": "numeric",
            "n": "transactionAmount",
            "v": "300.0"
          },
          { # Tuple Attribute
            "t": "string",
            "n": "dataSource",
            "v": "Recharge"
          },
          { # Tuple Attribute
            "t": "string",
            "n": "feedName",
            "v": "Recharge"
          }
        ],
        "event": "EVNT01" # Value from 'Event Identifier' configured on UI
      }
    ]
  },
  "gateway": "InboundGateway" # Value from payloadGatewayValue
}
```

Unica Journey Configuration

Unica Journey

Downstream Kafka is a distributed event store and stream-processing platform that captures events and send them to Unica Journet. You can configure the drive.json file with the kafka parameters that are necessary to map event details and ensure they are sent to the right destination. Based on the set parameters, events will display the set parameter values in the drop-down of the events files.

In the drive.json file, go to the *eventConsumers* section and set the Unica Journey parameters to send the event details. The sample configuration of downstream Kafka publishing events to Unica Journey is shown below:

```
"eventConsumers": [
  {
    "className": "com.hcl.drive.communication.UnicaJourneyEventConsumer",
    "name": "UnicaJourneyEventConsumer",
    "parameters": [
      {
        "name": "payloadEntrySourceCode",
        "stringValue": "ES-TEST002"
      },
      {
```

```
        "name": "databaseServerIp",
        "stringValue": "127.0.0.1"
      },
      {
        "name": "databaseServerUser",
        "stringValue": "drive"
      },
      {
        "name": "databaseServerPassword",
        "stringValue": "6F2FA7F6B3CD1570F19F0A0B2636E033"
      },
      {
        "int32Value": 3306,
        "name": "databaseServerPort"
      },
      {
        "name": "databaseName",
        "stringValue": "drive_acme_core"
      },
      {
        "name": "dateTimeAttributeName",
        "stringValue": "ts"
      },
      {
        "name": "producerConfigFilePath",
        "stringValue": "${HCL_HOME}/../${HCL_SOLUTION}/etc/producer_config.properties"
      },
      {
        "name": "producerKrb5ConfigFilePath",
        "stringValue": "/etc/krb5.conf"
      },
      {
        "name": "producerTopic",
        "stringValue": "JOURNEY_QUALIFIED_TRANS_TOPIC"
      }
    ]
  }
]
```

1. *eventConsumers.className*: Name of the Java class which writes the events in the pre-defined format of the downstream system. For Unica Journey, the className must be set as "*com.hcl.drive.communication.UnicaJourneyEventConsumer*".
2. *eventConsumers.name*: Name of the downstream system to be assigned.
3. Parameters:
    a. Database related:
        i. *databaseServerIp*: Server IP address of the database to publish the downstream events.
        ii. *databaseServerUser*: Username of the database.
        iii. *databaseServerPassword*: Encrypted password of the database.
        iv. *databaseServerPort*: Port number of the database.
        v. *databaseName*: Name of the database.
    b. Kafka related:
        i. *producerConfigFilePath*: Path to the producer_config.properties file. This file will have the configuration/properties for connecting to Kafka.

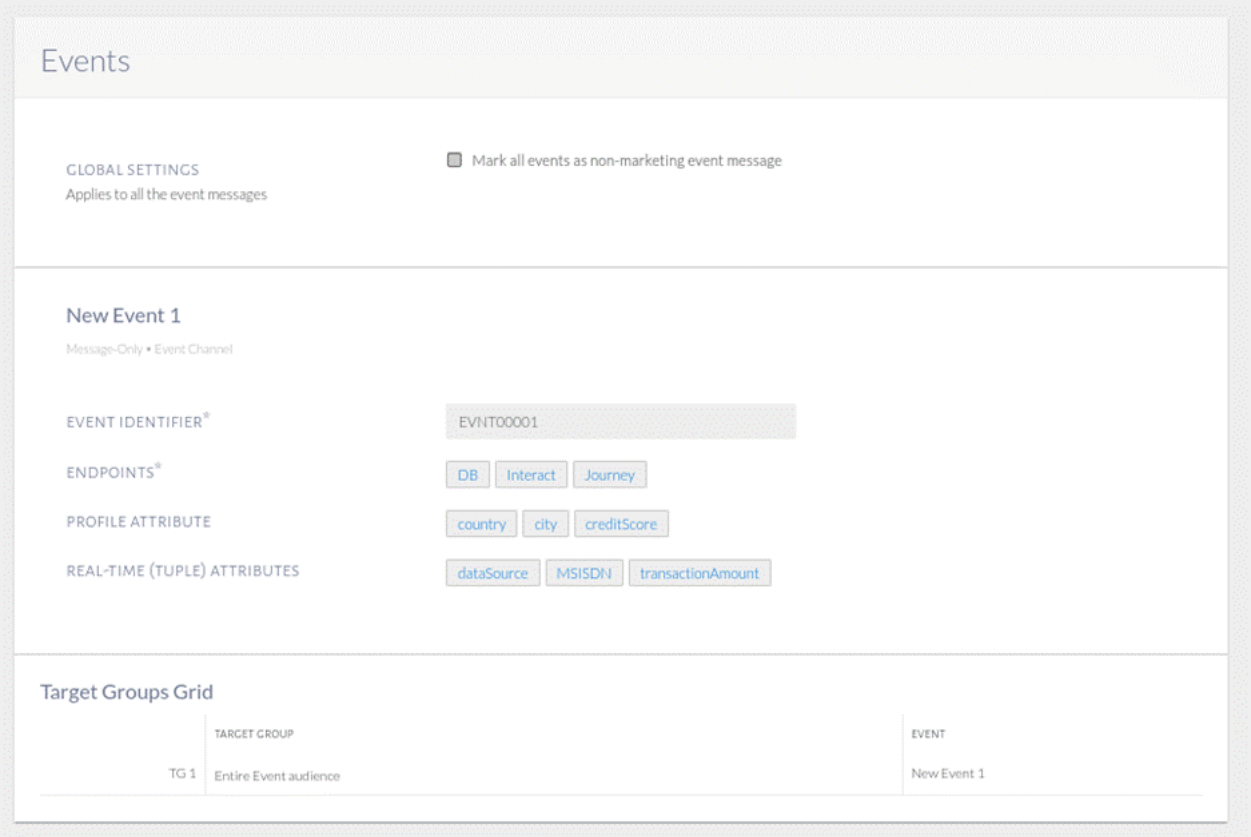
ii. *producerKrb5ConfigFilePath*: Path to the krb5.conf file if needed to Kafka connection needs to be Kerberos authenticated.

iii. *producerTopic*: Name of the Kafka Topic on which events will be published, and Unica Journey will read it from.

c. *payloadEntrySourceCode*: Value of the "entrySourceCode" to be populated when pushing the events to Unica Journey.

Sample Message

A sample screenshot shown below within an event configured in Detect where we select the attributes and populate an event identifier to push to downstream kafka topic for Unica Journey to ingest.



Sample message output to the Kafka topic based on the configuration above.

```
{
  "entrySourceCode": "ES-00000363", # Value from payloadEntrySourceCode
  "data": [
    {
      "eventID": "EVNT00001",        # Value configured from UI
      "country": "India",            # Profile Attribute
      "creditScore": "700",          # Profile Attribute
      "MSISDN": "9881153226",        # Tuple Attribute
      "city": "Mumbai",              # Profile Attribute
      "transactionAmount": "600.0",  # Tuple Attribute
      "dataSource": "Recharge"       # Tuple Attribute
```

```
    }
  ]
}
```

Chapter 5. Solution Source Code

A solution is a domain-specific configuration of Drive that is customized for a particular customer in that domain. Different solutions can be in different domains E.g., Telco, banking, retail, healthcare. They can also be tailored towards different customers in that domain E.g., different data sources, communication, and fulfillment services, etc.

To build custom solutions based on the requirement, the source code for it would reside here. All feed applications, its parsers and event consumers development would happen here. The enrichment helpers logic are written here after they are declared in the `enrichment_functions.json` file. Let's dive into each topic.

Feed Applications

A feed application consist of a flow with a defined set of operators in a certain structure. Operators are individual components designed to perform a certain set of tasks.

A typical feed application has the following components:

- A *Source*: receives the input data from an external data source
- A *Parser*: parses the input data into tuple format
- An *Enricher*: enriches the tuple data with profile information stored in PinPoint, and the aggregate data stored in FastPast
- An *Aggregator*: updates the aggregate data stored in FastPast
- A *Trigger Evaluator*: Detects trigger conditions of interest
- One or more Sinks: Sends trigger results to the campaign executor via a Kafka queue

A feed application is typically configured via parameters specified in `drive.json`. A typical entry would look something like this:

```
{"drive":{..."feedApplications":[{"logLevel":"INFO","name":"CC Usage"}]...}}
```

The source data for a feed application commonly come from files or Kafka queues. Custom source operators can be written for connecting to different sources.

An example of a simple feed application which would scan a directory for csv every 10 secs and ingest data would look like the code below:

```
classCCUsageFeedApplication(FeedApplication):"""Implements the Goliath
 feed application"""defcreate_flow(self):"""Creates the Credit Card Usage
 flow"""scan_schema=Schema({"filename":String,"modTime":Int64},name="scan_schema")parser_schema=Sche
ma({drive_constants.CA_EPOCH_ATTRIBUTE:Int64,drive_constants.CA_TIMESTAMP_ATTRIBUTE:Int64,"departmen
tName":String,"description":String,"cardHolderName":String,"customerID":String,"merchant":String,"ca
tegory":String,"mcc":String,"transactionAmount":Double,"transactionType":String},name="parser_schema
")scanner=self.instantiate_directory_scanner(file_name_filter=PatternBasedFileNameFilter(r".*\.csv")
,output_schema=scan_schema,period_in_seconds=10)parser=self.instantiate_operator(class_=CCUsageReade
r,input_schema=scan_schema,move_path=self.feed_data_move_dir(),name=drive_constants.DRIVE_APPLICATIO
N_PARSER_OPERATOR_NAME,output_schema=parser_schema)returnself.compose_flow(ingress_flow=scanner>>par
ser)
```

- The class `CCUsageFeedApplication` is inherited from `FeedApplication` base class.
- `create_flow` is an abstract method of `FeedApplication` class which is implemented here. This method is used to define the flow of the operators for the feed application.
- Every operator by default requires a 3 parameters: name, input schema and output schema.
- *Schema* defines the name and data type of the attributes that the operator would be receiving or sending.
- *Input Schema* defines the schema that would be entering the operator.
- *Output Schema* defines the schema that would be sent ahead from the operator.
- `scan_schema` and `parser_schema` are two such schema defined in the example above. `scan_schema` contains the attributes would be sent by the directory scanner operator which would be file name and the last modified time-stamp of the file. `parser_schema` would contain the attributes that would be populated after reading the input file from the source directory.
- `CCUsageReader` operator class will parse the contents of the file, raise errors if any, populate the output tuple and then emit it to the next operator.
- `compose_flow` adds on the standard pre-defined operators like aggregator, enricher and kafka sink after the ingress flow.

Parsers

Parser operators are used for parsing data from input source and setting it into tuple format. Here the data read from source is converted to a format which would be easier for the feed application to consume and process. A sample of a parser class operator would look like the snippet below:

```
classCCUsageReader(CSVParser):"""A sample operator that generates cc_usage related
 data"""CURRENCY_SIGN="$"@oxygen_operator()def__init__(self,batch_size=None,delimiter=",",move_path=
None):super(CCUsageReader,self).__init__(batch_size=batch_size,date_format=None,delimiter=delimiter,
epoch_attribute=CA_EPOCH_ATTRIBUTE,move_path=move_path,row_processor=self._parse_row)defset_up(self)
:super(CCUsageReader,self).set_up()def_parse_row(self,fields,out_tuple):"""Parses the fields in a
 row"""out_tuple.departmentName=fields[3].upper()out_tuple.cardHolderName=fields[4].upper()out_tuple
.customerID=fields[2]out_tuple.description=""out_tuple.merchant=fields[5].upper()out_tuple.category=
category=fields[6].upper()setattr(out_tuple,CA_TIMESTAMP_ATTRIBUTE,int(self._datetime_parser.utime(f
ields[7])))out_tuple.transactionAmount=float(fields[8].split(CCUsageReader.CURRENCY_SIGN)[1])out_tup
le.transactionType=fields[9]
```

- The class `CCUsageReader` is inherited from `CSVParser` base class.
- The `delimiter` passed to the constructor will split each line in the file and then send the list to the `_parse_row` method.
- The method `_parse_row` is passed on as `row_processor` to the constructor of the `CSVParser` class.
- The method will have 2 parameters passed to it - `fields` is the list of the raw data of a single line read from a file, the list is created based on the split set by the `delimiter`. `out_tuple` is the output tuple for the operator based on the output schema set in the feed application class.

Chapter 6. Detect Expression Language

Introduction

The HCL Detect Expression Language is a simple expression language used to specify filter conditions and basic arithmetic manipulations. It can be used as part of configuring triggers. In particular, a trigger's `WHEN` condition can have a free-form expression specified as part of a predicate's right hand-side. Similarly, a trigger's `THEN` action can have an expression specified as an assignment for an output attribute. These expressions are specified in HCL Detect Expression Language.

Types

An expression in HCL Detect Expression Language can have one of the following primitive types: `String` (a string), `Bool` (a Boolean), `Int16` (a 16-bit integer), `Int32` (a 32-bit integer), `Int64` (a 64-bit integer), or `Double` (a double precision floating point number). It can also have a list type, where the element type of the list is one of the primitive types: `List(String)`, `List(Bool)`, `List(Int16)`, `List(Int32)`, `List(Int64)`, `List(Double)`.

Literals

`Bool`, `Double`, `Int32`, `Int64`, and `String` literals can be present in an expression.

- A `Bool` literal can be either `true` or `false`.
- A `Double` literal is a decimal number that either contains the decimal separator (e.g., `3.5`, `.5`, `3.`) or is given in the scientific notation (e.g., `1e-4`, `1.5e3`). A *Double* literal must be between `(2 - 2 ^ -52) * 2 ^ 1023` (`~ 1.79 * 10 ^ 308`) and `-(2 - 2 ^ -52) * 2 ^ 1023` (`~ -1.79 * 10 ^ 308`). Moreover, the magnitude of a literal cannot be less than `2 ^ -1074` (`~ 4.94 * 10 ^ -324`). Furthermore,
    - a `NaN` (not a number) value can be specified through the `Double("nan")` expression,
    - an `Inf` (positive infinity) value can be specified through the `Double("inf")` expression,
    - and a `-Inf` (negative infinity) value can be specified through the `Double("-Inf")` expression.
- Integer literals without a suffix are of the `Int32` type (e.g., `14`). An `Int32` literal must be between `2 ^ 31 - 1` (`= 2147483647`) and `-2 ^ 31` (`= -2147483648`).
- The `L` suffix is used to create `Int64` literals (e.g., `14L`). An `Int64` literal must be between `2 ^ 63 - 1` (`= 9223372036854775807L`) and `-2 ^ 63` (`= -9223372036854775808L`).
- A `String` literal appears within double quotes, as in `"I'm a string literal"`. The escape character `\` can be used to represent new line (`\n`), tab (`\t`), double quote (`\"`) characters, as well as the escape character itself (`\\`).

Arithmetic Operations

An expression in HCL DetectExpression Language can contain the basic arithmetic operations: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulo (`%`) with the usual semantics. These are binary operations that expect sub-expressions on each side. An expression in HCL Detect Expression Language can also contain the unary minus (`-`) operation that expects a sub-expression on the right. Finally, parentheses (`()`) are used for adjusting precedence, as usual.

Addition operation corresponds to concatenation for the `String` type and is the only available operation on this type. `Bool` type does not support arithmetic operations. Division applies integer division on integer types and floating point division on `Double` types.

The modulo operation yields the remainder from the division of the first operand by the second. It always yields a result with the same sign as its second operand. The absolute value of the result is strictly smaller than the absolute value of the second operand.

Examples:

- A simple arithmetic expression: `(3+4*5.0)/2`
- A simple expression involving a `String` literal: `"area code\tcountry"`
- A unary minus expression: `-(3+5.0)`

Comparison Operations

An expression in HCL Detect Expression Language can contain the basic comparison operations: greater than (`>`), greater than or equal (`>=`), less than (`<`), less than or equal (`<=`), equals (`==`), and not equals (`!=`) with the usual semantics. These are binary operations. With the exception of equals and not equals, they expect sub-expressions of numerical types or strings on each side. For strings, the comparison is based on lexicographic order. For equals and not equals, the left and the right sub-expressions must be of compatible types with respect to HCL Detect Expression Language coercion rules. The result of the comparison is always of type `Bool`.

Examples:

- An expression using comparisons: `3>5`
- An expression using String comparisons: `"abc"<"def"`
- An expression using inequality comparison: `"abc"!="def"`
- An expression comparing a `Double` literal with a `NaN` (not a number) value: `10.5 != Double("nan")` (Note that in this case, the `math.isNaN` built-in function can also be used.)
- An expression comparing a `Double` literal with an `Inf` (positive infinity) value: `10.5 != Double("inf")` (Note that in this case, the `math.isPositiveInfinity` built-in function can also be used.)
- An expression comparing a `Double` literal with a `-Inf` (negative infinity) value: `10.5 != Double("-inf")` (Note that in this case, the `math.isNegativeInfinity` built-in function can also be used.)

Logical Operations

An expression in HCL Detect Expression Language can contain the basic logical operations: logical and (`&&`) and logical or (`||`) with the usual semantics. These are binary operations that expect sub-expressions of type `Bool` on each side. Shortcut is used to evaluate the logical operations. For logical *and*, if the sub-expression on the left evaluates to `false`, then the sub-expression on the right is not evaluated and the result is `false`. For logical *or*, if the sub-expression on the left evaluates to `true`, then the sub-expression on the right is not evaluated and the result is `true`. An expression in HCL Detect Expression Language can also contain the *not* (`!`) operator, which is a unary operator that expects a sub-expression of type `Bool` on the right.

Examples:

- A simple logical expression: `3>5||2<4`
- A logical expression that uses not: `!(3>5)`

Ternary Operation

An expression in HCL Detect Expression Language can make use of the ternary operation: `condition? choice1:choice2`. The condition of the ternary operation is expected to be of type `Bool` and the two choices are expected to be sub-expressions with compatible types. The ternary operation is lazily evaluated. If the condition evaluates to `true`, then the result is the first choice, without the sub- expression for the second choice being evaluated. If the condition evaluates to `false`, then the result is the second choice, without the sub-expression for the first choice being evaluated.

Examples:

- A simple ternary operation: `3<10?"smallerThan10":"notSmallerThan10"`

List Operations

A list is constructed by specifying a sequence of comma (`,`) separated values of the element type, surrounded by brackets (`[]`) . For instance, an example literal for `List(Double)` is `[3.5, 6.7, 8.3]` and an example for `List(String)` is `["HCL", "Detect"]`. An empty list requires casting to define its type. As an example, an empty `List(String)` can be specified as `List(String)([])`.

An expression in HCL Detect Expression Language can contain a few basic list operations: containment (`in`), non-containment (`not in`), indexing (`[i]`), slicing (`[i:j]`), concatenation (`+`), and size inquiring (`size()`).

Containment yields a Boolean answer, identifying whether an element is contained within a list. For instance, `3in[2,5,3]` yields `true`, whereas `8 in [2, 5, 3]` yields `false`. Non-containment is the negated version of the containment. For instance, `3 not in [2, 5, 3]` yields `false`, whereas `8 not in [2, 5, 3]` yields `true`.

Indexing yields the element at the specified index within the list. `0`-based indexing is used and indices are of type `Int32`. For instance, `[2, 5, 3][1]` yields `5`, and `[2, 5, 3][-3]` yields `2`. The index should be between `-size` and `size-1`, inclusive. An index that is out of these bounds will result in an evaluation error at runtime.

Slicing yields a sub-list. The start index is inclusive, whereas the end index is exclusive. If the range is out of bounds, then an empty list is returned. For instance, `[2,5,3,7][1:2]` yields `[5]`, `[2, 5, 3, 7][1:3]` yields `[5, 3]`, `[2, 5, 3, 7][1:1]` yields `[]`, `[2, 5, 3, 7][3:5]` yields `[7]`, `[2, 3, 4][-2:-1]` yields `[3]`, `[2, 3, 4][-5:-1]` yields `[2, 3]`, `[2, 3, 4][1, 6]` yields `[3, 4]`, and `[2, 5, 3, 7][4:6]` yields `[]`.

Concatenation results in a list that contains the elements from the first list followed by the elements from the second list. For instance, `[1,2]+[3]` yields `[1, 2, 3]`.

The size of a list can be retrieved via the built-in function `list.size()`.

Precedence and Associativity of Operations

Operations	Associativity
`()`	
`[]`	
unary `-`, `!`	
`*`, `/`, `%`	left
`+`, `-`	left
`>`, `>=`, `<`, `<=`	left
`==`, `!=`	left
`in`	
`&&`	left
`||`	left
`?:`	

Attributes

Expressions in HCL Detect Expression Language can also contain *attributes*. Attributes are identifiers that correspond to the attributes available in a tuple or in the master profile. Each attribute has a type and can appear in anywhere a sub-expression of that type is expected.

If an attribute comes from the master profile, it can be referenced by prefixing it with `profile.`. Otherwise, only the attribute name is used.

Examples:

- A string formed by concatenating a `String` literal and an attribute named `code` from the current tuple: `"AREA_" + code`
- A string formed by concatenating a `String` literal and a profile attribute named `name`: `"My name is " + profile.name`
- An arithmetic expression involving an attribute and `Int32` literals: `numSeconds / (24 * 60 * 60)`
- An arithmetic expression involving a profile attribute and `Int32` literals: `profile.ageInSeconds / (24 * 60 * 60)`
- A floating-point arithmetic expression, where the floating point literal is of type `Double`: `cost / 1000.0`
- Another expression involving a profile attribute, where the floating point literal is of type `Double`: `profile.revenue / 1000.0`
- A Boolean expression checking if a `String` literal is found in an attribute named `places` of type `List(String)`: `"airport" in places`
- A Boolean expression checking if a `String` literal is found in a profile attribute named `favoritePlaces` of type `List(String)`: `"airport" in profile.favoritePlaces`

Conversions

HCL Detect Expression Language supports explicit conversions via casts using a function call syntax. The name of the function is the name of the type we want to cast to.

Examples:

- Casting an `Int32` to a `String`: `String(14)` yields `"14"`
- Casting a `String` to a `Double`: `Double("4.5")` yields `4.5`
- Casting a `String` to a `Double`: `Double("nan")` yields `NaN` (not a number)

HCL Detect Expression Language supports implicit conversions (aka coercions) as well. When two integers of different types are involved in an operation, the one that has the smaller number of bits is coerced into the wider type. When an integer is involved in an operation with a `Double` it is coerced into a `Double`. Also note that, in such operations, `Int64` values that cannot be represented exactly will be rounded to the closest `Double` value.

Examples:

- Coercion with integers of different bit-lengths: `count/2` has type `Int64`, assuming `count` is of type `Int64` (this has the same semantics as the expression `count / Int64(2)`)
- Coercion involving an `Int32` and a `Double`: `timestamp / 1000` has type `Double`, assuming `timestamp` is of type `Double` (this has the same semantics as the expression `timestamp / Double(1000)`)

Aggregates

Expressions in HCL Detect Expression Language can also contain *aggregates*. Aggregates are summary statistics maintained at different temporal granularities. They are specified using their names, zero or more group by attributes (coming from the tuple being processed), period and the window unit.

The available periods are `Current`, `Last`, and `AllTime`. When the period is `Current`, the available window units are: `Day`, `Hour`, `Month` and `Year`. If the period is `Last`, the `Minute` can also be used as the window unit. The computed aggregates are always of type `Double`.

The following examples illustrate the use of aggregates as part of expressions:

- This aggregate, named `numCallsMade`, returns the number of calls made for a given number within the last hour, where `callingNumber` is an attribute available from the current tuple: `aggregate(numCallsMade[callingNumber], Last, 1, Hour)`.
- Aggregates can be involved in arithmetic operations as usual: `aggregate(numCallsMade[callingNumber],Last,1,Hour)+aggregate(numCallsMade[calledNumber], Last, 1, Hour)`
- Some aggregates may have no group by attributes: `aggregate(totalCalls,Current,Month)`
- Some aggregates may have multiple group by attributes: `aggregate(numCallsMade[callingNumber,callingCellTower],Last,1,Hour)`

Aggregates can also specify the number of most recent time units to be used for the aggregation. By default an hourly aggregate is computed from 6 10-minute aggregates, a daily aggregate is computed from 24 hourly aggregates, a monthly aggregate is computed from daily aggregates within a month, and a yearly aggregate is computed from 12 monthly aggregates. One can specify a second argument as part of the aggregate's temporal access function, which represents the number of time units to be used for the aggregation. The time units used are always the most recent ones. For instance:

- The following aggregate gets the number of calls made during the last week (7 days):
  `aggregate(numCallsMade[callingNumber],Last,7,Day)`

The aggregates with the `Current` and `AllTime` periods provide exact results:

- `aggregate(<aggregate>,Current,Hour)`: an exact aggregate value over all the activities within the current hour. E.g.: If the current time is 14:20pm, then the activities within the last 20 minutes are included.
- `aggregate(<aggregate>,Current,Day)`: an exact aggregate value over all the activities within the current day. E.g.: If the current time is 14:20pm, then the activities since midnight are included.
- `aggregate(<aggregate>,Current,Month)`: an exact aggregate value over all the activities with the current month. E.g.: If the current time is 14 May 14:20pm, then the activities since the beginning of May up to 14:20pm on May 14th are included.
- `aggregate(<aggregate>,Current,Year)`: an exact aggregate value over all the activities with the current year. E.g.: If the current time is 14 May 2017 14:20pm, then the activities since the beginning of 2017 up to 14:20pm on May 14th are included.
- `aggregate(<aggregate>,AllTime)`: an exact aggregate value over all the activities, irrespective of time.

There are 4 possible ways of computing aggregates with the `Last` period:

- **Aggregate over the last hour**: an approximate aggregate value over the activities within the last 60 minutes, that is the last six 10-minute periods. It is an approximate value in the sense that if the current 10-minute interval is at least half past, then the aggregate is over the activities within the current 10-minute interval plus the last five 10-minute intervals. If the current 10 minute interval is less than half past, then the aggregate is over the activities within the current 10-minute interval plus the last six 10-minute intervals. E.g.: If the current time is 14:29pm, then the activities within the interval [13:30pm - 14:29pm] are included. If the current time is 14:21pm, then the activities within the interval [13:20pm - 14:21pm] are included.
- **Aggregate over the last day**: an approximate aggregate value over the activities within the last 24 hours. It is an approximate value in the sense that if the current hour is at least half past, then the aggregate is over the activities within the current hour plus the last 23 calendar hours. If the current hour is less than half past, then the aggregate is over the activities within the current hour plus the last 24 calendar hours. E.g.: If the current time is Tuesday 14:50pm, then the activities within the interval [Monday 15:00pm - Tuesday 14:50pm] are included. If the current time is Tuesday 14:10pm, then the activities within the interval [Monday 14:00pm - Tuesday 14:10pm] are included.
- **Aggregate over the last month**: an approximate aggregate value over the activities within the last 30 days. It is an approximate value in the sense that if the current day is at least half past, then the aggregate is over the activities within the current day plus the last 29 calendar days. If the current day is less than half past, then the aggregate is over the activities within the current day plus the last 30 calendar days. E.g.: If the current time is 14 May 22:00pm,

then the activities within the interval [15 April 00:00am - 14 May 22:00pm] are included. If the current time is 14 May 02:00am, then the activities within the interval [14 April 00:00am - 14 May 02:00am] are included.
- **Aggregate over the last year**: an approximate aggregate value over the activities within the last 12 months. It is an approximate value in the sense that if the current month is at least half past, then the aggregate is over the activities within the current month plus the last 11 calendar months. If the current month is less than half past, then the aggregate is over the activities within the current month plus the last 12 calendar months. E.g.: If the current time is 28 May 2017 14:00pm, then the activities within the interval [1 June 00:00am - 28 May 14:00pm] are included. If the current time is 2 May 14:00pm, then the activities within the interval [1 May 00:00am - 2 May 14:00pm] are included.

The same kind of approximation applies if the number of most recent time units are specified while accessing an aggregate. For instance, if the last 4 days are requested from the last month, then the current day plus the last 3 or 4 calendar days are included in the result, depending on whether the current day is at least half past or not, respectively.

These computations are done by using the following aggregate expressions:

- `aggregate(<aggregate>,Last,<windowLength>,Minute)`: this computes the aggregation by using *windowLength* minutes from the last hour.
- `aggregate(<aggregate>,Last,<windowLength>,Hour)`: if *windowLength* is greater than 1, this computes the aggregation over the last *windowLength* hours from the last day. Otherwise it computes the aggregation using the last hour.
- `aggregate(<aggregate>,Last,<windowLength>,Day)`: if *windowLength* is greater than 1, this computes the aggregation over the last *windowLength* days from the last month. Otherwise it computes the aggregation over the last day.
- `aggregate(<aggregate>,Last,<windowLength>,Month)`: if *windowLength* is greater than 1, this computes the aggregation over the last *windowLength* months from the last year. Otherwise it computes the aggregation over the last month.
- `aggregate(<aggregate>,Last,<windowLength>,Year)`: this computes the aggregation over the last year and the *windowLength* must be equal to 1.

Null Values

Attributes in HCL Detect Expression Language are nullable, that is, attributes can take null values. To denote a null value, the `null` keyword is used.

Only the following actions are legal on the expressions with null values:

- **Built-in function calls**: A nullable function parameter can take a null value. E.g.: the second parameter in the `list.indicesOf([1, 2, null, 4, 5, null], null)` call
- **Ternary operations**: The values returned from choices can be null. E.g.: `string.startsWith(profile.areaCode, "AREA_") ? profile.areaCode : null` where `areaCode` is a profile attribute of the `String` type
- **List items**: Null values can be list items. E.g.: `[null, 3, 5, 6, null]`
- **List containment check operations**: Null values can be used on the left hand side. E.g.: `null not in [null, 3, 5, 6, null]`

- **Equality check operations**: Null values can be compared for equality and non-equality. E.g.: `MSISDN == null` where `MSISDN` is a tuple attribute (Note: For comparisons with null values, the `isNull` operator can also be used. Examples: `isNull(null)` yields `trueisNull(profile.x)` yields `false` if the `x` profile attribute is not null)
- **Type conversions**: The type conversion rules are similar to the ones for non-null expressions. Some examples that are legal: `List(Int32)(null)`, `Int32(null)`, `String(Int32(null))`, some examples that are illegal: `List(Int32)(List(Int64)(null))`, `Int64(List(Int32)(null))`, `Bool(Int32(null))`, `List(Int16)(Double(null))`

Chapter 7. Detect Expression Language Built-in Functions

Geohash Functions

geohash.covers

Bool geohash.covers(String geohash1, String geohash2)

Checks whether the first geohash covers the second one, where the two geohashes being equal is also considered a positive result.

Parameters:

- geohash1 - the first geohash (non-nullable).
- geohash2 - the second geohash (non-nullable).

Returns:

true if the first geohash covers the second one.

geohash.encode

String geohash.encode(Double latitude, Double longitude, Int32 level)

Encodes a geohash from latitude and longitude information.

Parameters:

- latitude - the latitude (non-nullable).
- longitude - the longitude (non-nullable).
- level - the level (non-nullable).

Returns:

the encoded geohash.

geohash.intersects

Bool geohash.intersects(String geohash1, String geohash2)

Checks whether the two geohashes intersect.

Parameters:

- geohash1 - the first geohash (non-nullable).
- geohash2 - the second geohash (non-nullable).

Returns:

true if the two geohashes intersect, false otherwise.

geohash.intersectsAny

Bool geohash.intersectsAny(List(String) geohashes1, List(String) geohashes2)

Checks whether any pair of geohashes from the two lists intersect.

Parameters:

- geohashes1 - the first geohash list (non-nullable, null elements not allowed).
- geohashes2 - the second geohash list (non-nullable, null elements not allowed).

Returns:

true if any pair of geohashes from the two lists intersect, false otherwise.

List Functions

list.average

<Numeric T> Double list.average(List(T) values)

Returns the average of the input values. If the list of values is empty, the operation yields a runtime error.

Parameters:

- values - the input values (non-nullable, null elements not allowed).

Returns:

the average value.

list.containsAll

<Primitive T> Bool list.containsAll(List(T) list, List(T) values)

Checks whether all of the values are in the input list.

Parameters:

- list - the input list (non-nullable, null elements allowed).
- values - the list of values to check (non-nullable, null elements allowed).

Returns:

true if all of the values are in the input list, false otherwise.

list.containsAny

<Primitive T> Bool list.containsAny(List(T) list, List(T) values)

Checks whether any one of the values are in the input list.

Parameters:

- list - the input list (non-nullable, null elements allowed).
- values - the list of values to check (non-nullable, null elements allowed).

Returns:

true if any one of the values are in the input list, false otherwise.

list.difference

<Primitive T> List(T) list.difference(List(T) list1, List(T) list2)

Returns the difference of the first input list from the second input list by preserving the insertion order of values in the first list and removing duplicates.

Parameters:

- list1 - the first list (non-nullable, null elements allowed).
- list2 - the second list (non-nullable, null elements allowed).

Returns:

the difference of the first list from the second one where the order of values in the first list is preserved and duplicates are removed.

list.disjoint

<Primitive T> Bool list.disjoint(List(T) list1, List(T) list2)

Checks whether the two input lists are disjoint.

Parameters:

- list1 - the first list (non-nullable, null elements allowed).
- list2 - the second list (non-nullable, null elements allowed).

Returns:

true if the input lists are disjoint, false otherwise.

list.indicesOf

<Primitive T> List(Int32) list.indicesOf(List(T) list, T value)

Finds the indices at which the value is present in the input list.

Parameters:

- list - the input list (non-nullable, null elements allowed).
- value - the value (nullable).

Returns:

the list of the indices at which the value is present in the input list.

list.intersection

<Primitive T> List(T) list.intersection(List(T) list1, List(T) list2)

Returns the intersection of the two lists by preserving the order of values in the first input list and removing duplicates.

Parameters:

- list1 - the first list (non-nullable, null elements allowed).
- list2 - the second list (non-nullable, null elements allowed).

Returns:

the intersection of the two lists where the order of values in the first list is preserved and duplicates are removed.

list.lookup

<Primitive T, Primitive R> R list.lookup(T key, List(T) key_list, List(R) value_list)

Performs a dictionary look-up of the given key in the key list then returns the matching value from the value list, produces a runtime error if the key is not found.

Parameters:

- key - the key (non-nullable).
- key_list - the key list (non-nullable, null elements not allowed).
- value_list - the value list (non-nullable, null elements allowed).

Returns:

the result of the dictionary look-up.

list.max

<Primitive T> T list.max(List(T) values)

Finds the maximum element of the input list, results in a runtime error if the list is empty.

Parameters:

- values - the input list (non-nullable, null elements not allowed).

Returns:

the maximum element of the input list.

list.min

<Primitive T> T list.min(List(T) values)

Finds the minimum element of the input list, results in a runtime error if the list is empty.

Parameters:

- values - the input list (non-nullable, null elements not allowed).

Returns:

the minimum element of the input list.

list.populationVariance

<Numeric T> Double list.populationVariance(List(T) values)

Returns the population variance of the input values. If the list of values is empty, the operation yields a runtime error.

Parameters:

- values - the input list (non-nullable, null elements not allowed).

Returns:

the population variance of the elements of the input list.

list.reverse

<Primitive T> List(T) list.reverse(List(T) list)

Returns the reverse of the input list.

Parameters:

- list - the input list (non-nullable, null elements allowed).

Returns:

the reversed list.

list.sampleVariance

<Numeric T> Double list.sampleVariance(List(T) values)

Returns the sample variance of the input values. If the length of the list of values is less than or equal to 1, the operation yields a runtime error.

Parameters:

- values - the input list (non-nullable, null elements not allowed).

Returns:

the sample variance of the elements of the input list.

list.size

<Primitive T> Int32 list.size(List(T) list)

Returns the size of the input list.

Parameters:

- list - the input list (non-nullable, null elements allowed).

Returns:

the size of the input list.

list.sort

<Primitive T> List(T) list.sort(List(T) list)

Returns the sorted version of the input list in ascending order.

Parameters:

- list - the input list (non-nullable, null elements not allowed).

Returns:

the sorted list.

list.subList

<Primitive T> List(T) list.subList(List(T) list, Int32 startPosition, Int32 endPosition)

Returns a slice of the input list. Gives a runtime error if the start or the end position is outside of its valid range.

Parameters:

- list - the input list (non-nullable, null elements allowed).
- startPosition - the start position of the slice (inclusive, in the range [-size,size]). A negative value indicates a position relative to the end of the list (non-nullable).
- endPosition - the end index of the slice (exclusive, in the range [-size,size]). A negative value indicates a position relative to the end of the list (non-nullable).

Returns:

a list representing the specified slice of the input.

list.sum

<Numeric T> Double list.sum(List(T) values)

Returns the sum of the input values. If the list of values is empty, the operation yields a runtime error.

Parameters:

- values - the input list (non-nullable, null elements not allowed).

Returns:

the sum of the input values.

list.union

<Primitive T> List(T) list.union(List(T) list1, List(T) list2)

Returns the union of the two lists by preserving the order of values in the first list and removing duplicates.

Parameters:

- list1 - the first list (non-nullable, null elements allowed).
- list2 - the second list (non-nullable, null elements allowed).

Returns:

the union of the two lists where the order of values in the first list is preserved and duplicates are removed.

Math Functions

math.ceil

Double math.ceil(Double value)

Returns the ceiling of the input value, i.e., the value of the smallest integer greater than or equal to the value.

Parameters:

- value - the input value (non-nullable).

Returns:

the ceiling of the input value.

math.floor

Double math.floor(Double value)

Returns the floor of the input value, i.e., the value of the largest integer greater than or equal to the value.

Parameters:

- value - the input value (non-nullable).

Returns:

the floor of the input value.

math.isInfinity

<Numeric T> Bool math.isInfinity(T value)

Checks whether the input value is infinity.

Parameters:

- value - the input value (non-nullable).

Returns:

true if the input value is infinity, false otherwise.

math.isNaN

<Numeric T> Bool math.isNaN(T value)

Checks whether the input value is NaN (not a number).

Parameters:

- value - the input value (non-nullable).

Returns:

true if the input value is NaN (not a number), false otherwise.

math.isNegativeInfinity

<Numeric T> Bool math.isNegativeInfinity(T value)

Checks whether the input value is negative infinity.

Parameters:

- value - the input value (non-nullable).

Returns:

true if the input value is negative infinity, false otherwise.

math.isPositiveInfinity

<Numeric T> Bool math.isPositiveInfinity(T value)

Checks whether the input value is positive infinity.

Parameters:

- value - the input value (non-nullable).

Returns:

true if the input value is positive infinity, false otherwise.

math.log

Double math.log(Double value)

Returns the natural logarithm of the input value.

Parameters:

- value - the input value (non-nullable).

Returns:

the natural logarithm of the input value.

math.max

<Primitive T> T math.max(T a, T b)

Finds the maximum of the two values.

Parameters:

- a - the first value (non-nullable).
- b - the second value (non-nullable).

Returns:

the maximum of the values.

math.min

<Primitive T> T math.min(T a, T b)

Finds the minimum of the two values.

Parameters:

- a - the first value (non-nullable).
- b - the second value (non-nullable).

Returns:

the minimum of the values.

math.pow

Double math.pow(Double base, Double exponent)

Returns the base value raised to the exponent.

Parameters:

- base - the base (non-nullable).
- exponent - the exponent (non-nullable).

Returns:

the base value raised to the exponent.

String Functions

string.indexOf

Int32 string.indexOf(String string, String substring, Int32 fromPosition)

Finds the index of the first occurrence of the sub-string within the input string, starting the search at a given start index. Gives a runtime error if the start or the end position is outside of its valid range.

Parameters:

- string - the input string (non-nullable).
- substring - the sub-string to be searched (non-nullable).
- fromPosition - the start position of the search (inclusive, in the range [-size,size]). A negative value indicates a position relative to the end of the string (non-nullable).

Returns:

the index of the sub-string, or -1 if not found.

string.join

<Primitive T> String string.join(List(T) values, String delimiter)

Returns a string that is the result of joining the string representations of the input list elements with the specified delimiter.

Parameters:

- values - the input list (non-nullable, null elements not allowed).
- delimiter - the string to be used as a delimiter (non-nullable).

Returns:

the joined string.

string.length

Int32 string.length(String string)

Returns the length of the input string.

Parameters:

- string - the input string (non-nullable).

Returns:

the length of the input string.

string.regexMatch

List(String) string.regexMatch(String string, String regex)

Finds all non-overlapping sub-strings of the input string that match the regular expression (regex).

Parameters:

- string - the input string (non-nullable).
- regex - the regex to be searched (non-nullable).

Returns:

the list of matching substrings.

string.split

List(String) string.split(String string, String separator)

Splits the list of sub-strings of the input string resulting from splitting it via the separator.

Parameters:

- string - the input string (non-nullable).
- separator - the separator string (non-nullable).

Returns:

the substrings.

string.startsWith

Bool string.startsWith(String input, String prefix)

Checks whether a string starts with a specified prefix.

Parameters:

- input - the string to check (non-nullable).
- prefix - the prefix (non-nullable).

Returns:

true if the string starts with the specified prefix, false otherwise.

string.substring

String string.substring(String string, Int32 startPosition, Int32 endPosition)

Creates a sub-string from the input string. Gives a runtime error if the start or the end position is outside of its valid range.

Parameters:

- string - the input string (non-nullable).
- startPosition - the start position of the sub-string (inclusive, in the range [-size,size]). A negative value indicates a position relative to the end of the string (non-nullable).
- endPosition - the end position of the sub-string (exclusive, in the range [-size,size]). A negative value indicates a position relative to the end of the string (non-nullable).

Returns:

the substring.

string.toLowerCase

String string.toLowerCase(String string)

Returns the lowercase representation of the string.

Parameters:

- string - the input string (non-nullable).

Returns:

the input string converted to lowercase.

string.toUpperCase

String string.toUpperCase(String string)

Returns the uppercase representation of the string.

Parameters:

- string - the input string (non-nullable).

Returns:

the input string converted to uppercase.

Time Functions

time.currentTimeInSeconds

Double time.currentTimeInSeconds()

Returns the current fractional seconds since the Epoch.

Parameters:

Returns:

the time since the Epoch.

time.daysToHours

<Integral T> Int64 time.daysToHours(T days)

Converts the time duration given in days to hours (as integer).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of hours (days x hours in a day = days x 24).

time.daysToMicros

<Integral T> Int64 time.daysToMicros(T days)

Converts the time duration given in days to microseconds (as integer).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of microseconds (days x (micros in a day) = days x (24 x 60 x 60 x 1000 x 1000)).

time.daysToMillis

<Integral T> Int64 time.daysToMillis(T days)

Converts the time duration given in days to milliseconds (as integer).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (days x (millis in a day) = days x (24 x 60 x 60 x 1000)).

time.daysToMinutes

<Integral T> Int64 time.daysToMinutes(T days)

Converts the time duration given in days to minutes (as integer).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of minutes (days x (minutes in a day) = days x (24 x 60)).

time.daysToNanos

<Integral T> Int64 time.daysToNanos(T days)

Converts the time duration given in days to nanoseconds (as integer).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (days x (nanos in a day) = days x (24 x 60 x 60 x 1000 x 1000 x 1000)).

time.daysToSeconds

<Integral T> Int64 time.daysToSeconds(T days)

Converts the time duration given in days to seconds (as integer).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of seconds (days x (seconds in a day) = days x (24 x 60 x 60)).

time.formatDateTime

String time.formatDateTime(String formatString, List(Int32) timeComponents)

Returns a formatted string including the given date.

Parameters:

- formatString - format string where the following specifiers are allowed:

%A : Full name of the day of the week (e.g., Tuesday)

%B : Full name of the month of the year (e.g., March)

%H : Two-digit 24-hour clock (in the range [00:23])

%I : Two-digit 12-hour clock (in the range [00:11])

%M : Minute within the hour (two-digits, in the range [00:59])

%S : Second within the minute (two-digits, in the range [00:59])

%Y : Year in four digits (in the range [1:9999])

%a : Short name of the day of the week (e.g., Tue)

%b : Short name of the month of the year (e.g., Mar)

%d : Day of the month (two-digits, in the range [01:31])

%j : Day of the year (three-digits, in the range [001:366])

%m : Month of the year (two-digits, in the range [01:12])

%p : Morning/afternoon marker (AM or PM)

%yLast two digits of the year (in the range [00:99])

(non-nullable).

- timeComponents - the date-time components of the form: [year, month, mday, hour, minute, second, wday, yday, isdst]

year: Year as a decimal number

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise (non-nullable, null elements not allowed).

Returns:

a formatted time string.

time.fractionalDaysToHours

<Numeric T> Double time.fractionalDaysToHours(T days)

Converts the time duration given in days to hours (in fractional form).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of hours (days x hours in a day = days x 24).

time.fractionalDaysToMicros

<Numeric T> Double time.fractionalDaysToMicros(T days)

Converts the time duration given in days to microseconds (in fractional form).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of microseconds (days x (micros in a day) = days x (24 x 60 x 60 x 1000 x 1000)).

time.fractionalDaysToMillis

<Numeric T> Double time.fractionalDaysToMillis(T days)

Converts the time duration given in days to milliseconds (in fractional form).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (days x (millis in a day) = days x (24 x 60 x 60 x 1000)).

time.fractionalDaysToMinutes

<Numeric T> Double time.fractionalDaysToMinutes(T days)

Converts the time duration given in days to minutes (in fractional form).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of minutes (days x (minutes in a day) = days x (24 x 60)).

time.fractionalDaysToNanos

<Numeric T> Double time.fractionalDaysToNanos(T days)

Converts the time duration given in days to nanoseconds (in fractional form).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (days x (nanos in a day) = days x (24 x 60 x 60 x 1000 x 1000 x 1000)).

time.fractionalDaysToSeconds

<Numeric T> Double time.fractionalDaysToSeconds(T days)

Converts the time duration given in days to seconds (in fractional form).

Parameters:

- days - the time duration given in days (non-nullable).

Returns:

the time duration expressed in terms of seconds (days x (seconds in a day) = days x (24 x 60 x 60)).

time.fractionalHoursToDays

<Numeric T> Double time.fractionalHoursToDays(T hours)

Converts the time duration given in hours to days (in fractional form).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of days (one 24th of hours).

time.fractionalHoursToMicros

<Numeric T> Double time.fractionalHoursToMicros(T hours)

Converts the time duration given in hours to microseconds (in fractional form).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of microseconds (hours x (micros in an hour) = hours x (60 x 60 x 1000 x 1000)).

time.fractionalHoursToMillis

<Numeric T> Double time.fractionalHoursToMillis(T hours)

Converts the time duration given in hours to milliseconds (in fractional form).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (hours x millis in an hour = hours x (60 x 60 x 1000)).

time.fractionalHoursToMinutes

<Numeric T> Double time.fractionalHoursToMinutes(T hours)

Converts the time duration given in hours to minutes (in fractional form).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of minutes (hours x minutes in an hour = hours x 60).

time.fractionalHoursToNanos

<Numeric T> Double time.fractionalHoursToNanos(T hours)

Converts the time duration given in hours to nanoseconds (in fractional form).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (hours x (nanos in an hour) = hours x (60 x 60 x 1000 x 1000 x 1000)).

time.fractionalHoursToSeconds

<Numeric T> Double time.fractionalHoursToSeconds(T hours)

Converts the time duration given in hours to seconds (in fractional form).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of seconds (hours x (seconds in an hour) = hours x (60 x 60)).

time.fractionalMicrosToDays

<Numeric T> Double time.fractionalMicrosToDays(T micros)

Converts the time duration given in microseconds to days (in fractional form).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of days (micros / (micros in a day) = micros / (24 x 60 x 60 x 1000 x 1000)).

time.fractionalMicrosToHours

<Numeric T> Double time.fractionalMicrosToHours(T micros)

Converts the time duration given in microseconds to hours (in fractional form).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of hours (micros / (micros in an hour) = micros / (60 x 60 x 1000 x 1000)).

time.fractionalMicrosToMillis

<Numeric T> Double time.fractionalMicrosToMillis(T micros)

Converts the time duration given in microseconds to milliseconds (in fractional form).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (micros / micros in a millisecond = micros / 1000).

time.fractionalMicrosToMinutes

<Numeric T> Double time.fractionalMicrosToMinutes(T micros)

Converts the time duration given in microseconds to minutes (in fractional form).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (micros / (micros in a minute) = micros / (60 x 1000 x 1000)).

time.fractionalMicrosToNanos

<Numeric T> Double time.fractionalMicrosToNanos(T micros)

Converts the time duration given in microseconds to nanoseconds (in fractional form).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (micros x nanos in a microsecond = micros x 1000).

time.fractionalMicrosToSeconds

<Numeric T> Double time.fractionalMicrosToSeconds(T micros)

Converts the time duration given in microseconds to seconds (in fractional form).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of seconds (micros / (micros in a second) = micros / (1000 x 1000)).

time.fractionalMillisToDays

<Numeric T> Double time.fractionalMillisToDays(T millis)

Converts the time duration given in milliseconds to days (in fractional form).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of days (millis / (millis in a day) = millis / (24 x 60 x 60 x 1000)).

time.fractionalMillisToHours

<Numeric T> Double time.fractionalMillisToHours(T millis)

Converts the time duration given in milliseconds to hours (in fractional form).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of hours (millis / (millis in an hour) = millis / (60 x 60 x 1000)).

time.fractionalMillisToMicros

<Numeric T> Double time.fractionalMillisToMicros(T millis)

Converts the time duration given in milliseconds to microseconds (in fractional form).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of microseconds (millis x micros in a millisecond = millis x 1000).

time.fractionalMillisToMinutes

<Numeric T> Double time.fractionalMillisToMinutes(T millis)

Converts the time duration given in milliseconds to minutes (in fractional form).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (millis / (millis in a minute) = millis / (60 x 1000)).

time.fractionalMillisToNanos

<Numeric T> Double time.fractionalMillisToNanos(T millis)

Converts the time duration given in milliseconds to nanoseconds (in fractional form).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (millis x (nanos in a millisecond) = millis x (1000 x 1000)).

time.fractionalMillisToSeconds

<Numeric T> Double time.fractionalMillisToSeconds(T millis)

Converts the time duration given in milliseconds to seconds (in fractional form).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of seconds (millis / millis in a second = millis / 1000).

time.fractionalMinutesToDays

<Numeric T> Double time.fractionalMinutesToDays(T minutes)

Converts the time duration given in minutes to days (in fractional form).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of days (minutes / (minutes in a day) = minutes / (24 x 60)).

time.fractionalMinutesToHours

<Numeric T> Double time.fractionalMinutesToHours(T minutes)

Converts the time duration given in minutes to hours (in fractional form).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of hours (minutes / minutes in an hour = minutes / 60).

time.fractionalMinutesToMicros

<Numeric T> Double time.fractionalMinutesToMicros(T minutes)

Converts the time duration given in minutes to microseconds (in fractional form).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of microseconds (minutes x (micros in a minute) = minutes x (60 x 1000 x 1000)).

time.fractionalMinutesToMillis

<Numeric T> Double time.fractionalMinutesToMillis(T minutes)

Converts the time duration given in minutes to milliseconds (in fractional form).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (minutes x (millis in a minute) = minutes x (60 x 1000)).

time.fractionalMinutesToNanos

<Numeric T> Double time.fractionalMinutesToNanos(T minutes)

Converts the time duration given in minutes to nanoseconds (in fractional form).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (minutes x (nanos in a minute) = minutes x (60 x 1000 x 1000 x 1000)).

time.fractionalMinutesToSeconds

<Numeric T> Double time.fractionalMinutesToSeconds(T minutes)

Converts the time duration given in minutes to seconds (in fractional form).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of seconds (minutes x seconds in a minute = minutes x 60).

time.fractionalNanosToDays

<Numeric T> Double time.fractionalNanosToDays(T nanos)

Converts the time duration given in nanoseconds to days (in fractional form).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of days (nanos / (nanos in a day) = nanos / (24 x 60 x 60 x 1000 x 1000 x 1000)).

time.fractionalNanosToHours

<Numeric T> Double time.fractionalNanosToHours(T nanos)

Converts the time duration given in nanoseconds to hours (in fractional form).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of hours (nanos / (nanos in an hour) = nanos / (60 x 60 x 1000 x 1000 x 1000)).

time.fractionalNanosToMicros

<Numeric T> Double time.fractionalNanosToMicros(T nanos)

Converts the time duration given in nanoseconds to microseconds (in fractional form).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of microseconds (nanos / nanos in a microsecond = nanos / 1000).

time.fractionalNanosToMillis

<Numeric T> Double time.fractionalNanosToMillis(T nanos)

Converts the time duration given in nanoseconds to milliseconds (in fractional form).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (nanos / (nanos in an millisecond) = nanos / (1000 x 1000)).

time.fractionalNanosToMinutes

<Numeric T> Double time.fractionalNanosToMinutes(T nanos)

Converts the time duration given in nanoseconds to minutes (in fractional form).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (nanos / (nanos in a minute) = nanos / (60 x 1000 x 1000 x 1000)).

time.fractionalNanosToSeconds

<Numeric T> Double time.fractionalNanosToSeconds(T nanos)

Converts the time duration given in nanoseconds to seconds (in fractional form).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of seconds (nanos / (nanos in a second) = nanos / (1000 x 1000 x 1000)).

time.fractionalSecondsToDays

<Numeric T> Double time.fractionalSecondsToDays(T seconds)

Converts the time duration given in seconds to days (in fractional form).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of days (seconds / (seconds in a day) = seconds / (24 x 60 x 60)).

time.fractionalSecondsToHours

<Numeric T> Double time.fractionalSecondsToHours(T seconds)

Converts the time duration given in seconds to hours (in fractional form).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of hours (seconds / (seconds in an hour) = seconds / (60 x 60)).

time.fractionalSecondsToMicros

<Numeric T> Double time.fractionalSecondsToMicros(T seconds)

Converts the time duration given in seconds to microseconds (in fractional form).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of microseconds (seconds x (micros in a second) = seconds x (1000 x 1000)).

time.fractionalSecondsToMillis

<Numeric T> Double time.fractionalSecondsToMillis(T seconds)

Converts the time duration given in seconds to milliseconds (in fractional form).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (seconds x millis in a second = seconds x 1000).

time.fractionalSecondsToMinutes

<Numeric T> Double time.fractionalSecondsToMinutes(T seconds)

Converts the time duration given in seconds to minutes (in fractional form).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (seconds / seconds in a minute = seconds / 60).

time.fractionalSecondsToNanos

<Numeric T> Double time.fractionalSecondsToNanos(T seconds)

Converts the time duration given in seconds to nanoseconds (in fractional form).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (seconds x (nanos in a second) = seconds x (1000 x 1000 x 1000)).

time.hoursToDays

<Integral T> Int64 time.hoursToDays(T hours)

Converts the time duration given in hours to days (as integer).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of days (one 24th of hours - integer division).

time.hoursToMicros

<Integral T> Int64 time.hoursToMicros(T hours)

Converts the time duration given in hours to microseconds (as integer).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of microseconds (hours x (micros in an hour) = hours x (60 x 60 x 1000 x 1000)).

time.hoursToMillis

<Integral T> Int64 time.hoursToMillis(T hours)

Converts the time duration given in hours to milliseconds (as integer).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (hours x millis in an hour = hours x (60 x 60 x 1000)).

time.hoursToMinutes

<Integral T> Int64 time.hoursToMinutes(T hours)

Converts the time duration given in hours to minutes (as integer).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of minutes (hours x minutes in an hour = hours x 60).

time.hoursToNanos

<Integral T> Int64 time.hoursToNanos(T hours)

Converts the time duration given in hours to nanoseconds (as integer).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (hours x (nanos in an hour) = hours x (60 x 60 x 1000 x 1000 x 1000)).

time.hoursToSeconds

<Integral T> Int64 time.hoursToSeconds(T hours)

Converts the time duration given in hours to seconds (as integer).

Parameters:

- hours - the time duration given in hours (non-nullable).

Returns:

the time duration expressed in terms of seconds (hours x (seconds in an hour) = hours x (60 x 60)).

time.localDateTimeFromDate

List(Int32) time.localDateTimeFromDate(Int32 year, Int32 month, Int32 day)

Produces the local date-time components ([year, month, mday, hour, minute, second, wday, yday, isdst]) according to the given date information (year, month and day)

year: Year as a decimal number

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

Parameters:

- year - year (in the range [1:9999]) (non-nullable).
- month - month (in the range [1:12]) (non-nullable).
- day - the day of the month (in the range [1:31]) (non-nullable).

Returns:

the local date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst].

time.localDateTimeFromDateTime

List(Int32) time.localDateTimeFromDateTime(Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second)

Produces the local date-time components ([year, month, mday, hour, minute, second, wday, yday, isdst]) according to the given date and time information (year, month, day, hour, minute and second)

year: Year as a decimal number

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

Parameters:

- year - the year (in the range [1:9999]) (non-nullable).
- month - the month within the year (in the range [1:12]) (non-nullable).
- day - the day within the month (in the range [1:31]) (non-nullable).
- hour - the hour within the day (in the range [0:23]) (non-nullable).

- minute - the minute within the hour (in the range [0:59]) (non-nullable).
- second - the second within the minute (in the range [0:59]) (non-nullable).

Returns:

the local date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst].

time.localTime

List(Int32) time.localTime(Double value)

Converts the time in seconds since the Epoch to the local date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst]

year: Year as a decimal

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

Parameters:

- value - the time (fractional seconds since the Epoch) (non-nullable).

Returns:

the local date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst].

time.microsToDays

<Integral T> Int64 time.microsToDays(T micros)

Converts the time duration given in microseconds to days (as integer).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of days (micros / (micros in a day) = micros / (24 x 60 x 60 x 1000 x 1000) - integer division).

time.microsToHours

<Integral T> Int64 time.microsToHours(T micros)

Converts the time duration given in microseconds to hours (as integer).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of hours (micros / (micros in an hour) = micros / (60 x 60 x 1000 x 1000) - integer division).

time.microsToMillis

<Integral T> Int64 time.microsToMillis(T micros)

Converts the time duration given in microseconds to milliseconds (as integer).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (micros / micros in a millisecond = micros / 1000 - integer division).

time.microsToMinutes

<Integral T> Int64 time.microsToMinutes(T micros)

Converts the time duration given in microseconds to minutes (as integer).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (micros / (micros in a minute) = micros / (60 x 1000 x 1000) - integer division).

time.microsToNanos

<Integral T> Int64 time.microsToNanos(T micros)

Converts the time duration given in microseconds to nanoseconds (as integer).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (micros x nanos in a microsecond = micros x 1000).

time.microsToSeconds

<Integral T> Int64 time.microsToSeconds(T micros)

Converts the time duration given in microseconds to seconds (as integer).

Parameters:

- micros - the time duration given in microseconds (non-nullable).

Returns:

the time duration expressed in terms of seconds (micros / (micros in a second) = micros / (1000 x 1000) - integer division).

time.millisToDays

<Integral T> Int64 time.millisToDays(T millis)

Converts the time duration given in milliseconds to days (as integer).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of days (millis / (millis in a day) = millis / (24 x 60 x 60 x 1000) - integer division).

time.millisToHours

<Integral T> Int64 time.millisToHours(T millis)

Converts the time duration given in milliseconds to hours (as integer).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of hours (millis / (millis in an hour) = millis / (60 x 60 x 1000) - integer division).

time.millisToMicros

<Integral T> Int64 time.millisToMicros(T millis)

Converts the time duration given in milliseconds to microseconds (as integer).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of microseconds (millis x micros in a millisecond = millis x 1000).

time.millisToMinutes

<Integral T> Int64 time.millisToMinutes(T millis)

Converts the time duration given in milliseconds to minutes (as integer).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (millis / (millis in a minute) = millis / (60 x 1000) - integer division).

time.millisToNanos

<Integral T> Int64 time.millisToNanos(T millis)

Converts the time duration given in milliseconds to nanoseconds (as integer).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (millis x (nanos in a millisecond) = millis x (1000 x 1000)).

time.millisToSeconds

<Integral T> Int64 time.millisToSeconds(T millis)

Converts the time duration given in milliseconds to seconds (as integer).

Parameters:

- millis - the time duration given in milliseconds (non-nullable).

Returns:

the time duration expressed in terms of seconds (millis / millis in a second = millis / 1000 - integer division).

time.minutesToDays

<Integral T> Int64 time.minutesToDays(T minutes)

Converts the time duration given in minutes to days (as integer).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of days (minutes / (minutes in a day) = minutes / (24 x 60) - integer division).

time.minutesToHours

<Integral T> Int64 time.minutesToHours(T minutes)

Converts the time duration given in minutes to hours (as integer).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of hours (minutes / minutes in an hour = minutes / 60 - integer division).

time.minutesToMicros

<Integral T> Int64 time.minutesToMicros(T minutes)

Converts the time duration given in minutes to microseconds (as integer).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of microseconds (minutes x (micros in a minute) = minutes x (60 x 1000 x 1000)).

time.minutesToMillis

<Integral T> Int64 time.minutesToMillis(T minutes)

Converts the time duration given in minutes to milliseconds (as integer).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (minutes x (millis in a minute) = minutes x (60 x 1000)).

time.minutesToNanos

<Integral T> Int64 time.minutesToNanos(T minutes)

Converts the time duration given in minutes to nanoseconds (as integer).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (minutes x (nanos in a minute) = minutes x (60 x 1000 x 1000 x 1000)).

time.minutesToSeconds

<Integral T> Int64 time.minutesToSeconds(T minutes)

Converts the time duration given in minutes to seconds (as integer).

Parameters:

- minutes - the time duration given in minutes (non-nullable).

Returns:

the time duration expressed in terms of seconds (minutes x seconds in a minute = minutes x 60).

time.nanosToDays

<Integral T> Int64 time.nanosToDays(T nanos)

Converts the time duration given in nanoseconds to days (as integer).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of days (nanos / (nanos in a day) = nanos / (24 x 60 x 60 x 1000 x 1000 x 1000) - integer division).

time.nanosToHours

<Integral T> Int64 time.nanosToHours(T nanos)

Converts the time duration given in nanoseconds to hours (as integer).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of hours (nanos / (nanos in an hour) = nanos / (60 x 60 x 1000 x 1000 x 1000) - integer division).

time.nanosToMicros

<Integral T> Int64 time.nanosToMicros(T nanos)

Converts the time duration given in nanoseconds to microseconds (as integer).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of microseconds (nanos / nanos in a microsecond = nanos / 1000 - integer division).

time.nanosToMillis

<Integral T> Int64 time.nanosToMillis(T nanos)

Converts the time duration given in nanoseconds to milliseconds (as integer).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (nanos / (nanos in an millisecond) = nanos / (1000 x 1000) - integer division).

time.nanosToMinutes

<Integral T> Int64 time.nanosToMinutes(T nanos)

Converts the time duration given in nanoseconds to minutes (as integer).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (nanos / (nanos in a minute) = nanos / (60 x 1000 x 1000 x 1000) - integer division).

time.nanosToSeconds

<Integral T> Int64 time.nanosToSeconds(T nanos)

Converts the time duration given in nanoseconds to seconds (as integer).

Parameters:

- nanos - the time duration given in nanoseconds (non-nullable).

Returns:

the time duration expressed in terms of seconds (nanos / (nanos in a second) = nanos / (1000 x 1000 x 1000) - integer division).

time.parseLocalDateTime

List(Int32) time.parseLocalDateTime(String dateTimeString, String formatString)

Produces the date-time components of a local date-time string according to the given format.

Parameters:

- dateTimeString - a local date-time string (non-nullable).
- formatString - a format string where the following specifiers are allowed:

%A : Full name of the day of the week (e.g., Tuesday)

%B : Full name of the month of the year (e.g., March)

%H : Two-digit 24-hour clock (in the range [00:23])

%I : Two-digit 12-hour clock (in the range [00:11])

%M : Minute within the hour (two-digits, in the range [00:59])

%S : Second within the minute (two-digits, in the range [00:59])

%Y : Year in four digits (in the range [1:9999])

%a : Short name of the day of the week (e.g., Tue)

%b : Short name of the month of the year (e.g., Mar)

%d : Day of the month (two-digits, in the range [01:31])

%j : Day of the year (three-digits, in the range [001:366])

%m : Month of the year (two-digits, in the range [01:12])

%p : Morning/afternoon marker (AM or PM)

%yLast two digits of the year (in the range [00:99])

(non-nullable).

Returns:

the date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst]

year: Year as a decimal

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

time.parseUtcDateTime

List(Int32) time.parseUtcDateTime(String dateTimeString, String formatString)

Produces the date-time components of a UTC date-time string according to the given format.

Parameters:

- dateTimeString - a UTC date-time string (non-nullable).
- formatString - a format string where the following specifiers are allowed:

%A : Full name of the day of the week (e.g., Tuesday)

%B : Full name of the month of the year (e.g., March)

%H : Two-digit 24-hour clock (in the range [00:23])

%I : Two-digit 12-hour clock (in the range [00:11])

%M : Minute within the hour (two-digits, in the range [00:59])

%S : Second within the minute (two-digits, in the range [00:59])

%Y : Year in four digits (in the range [1:9999])

%a : Short name of the day of the week (e.g., Tue)

%b : Short name of the month of the year (e.g., Mar)

%d : Day of the month (two-digits, in the range [01:31])

%j : Day of the year (three-digits, in the range [001:366])

%m : Month of the year (two-digits, in the range [01:12])

%p : Morning/afternoon marker (AM or PM)

%yLast two digits of the year (in the range [00:99])

(non-nullable).

Returns:

the date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst]

year: Year as a decimal

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

time.secondsToDays

<Integral T> Int64 time.secondsToDays(T seconds)

Converts the time duration given in seconds to days (as integer).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of days (seconds / (seconds in a day) = seconds / (24 x 60 x 60) - integer division).

time.secondsToHours

<Integral T> Int64 time.secondsToHours(T seconds)

Converts the time duration given in seconds to hours (as integer).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of hours (seconds / (seconds in an hour) = seconds / (60 x 60) - integer division).

time.secondsToMicros

<Integral T> Int64 time.secondsToMicros(T seconds)

Converts the time duration given in seconds to microseconds (as integer).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of microseconds (seconds x (micros in a second) = seconds x (1000 x 1000)).

time.secondsToMillis

<Integral T> Int64 time.secondsToMillis(T seconds)

Converts the time duration given in seconds to milliseconds (as integer).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of milliseconds (seconds x millis in a second = seconds x 1000).

time.secondsToMinutes

<Integral T> Int64 time.secondsToMinutes(T seconds)

Converts the time duration given in seconds to minutes (as integer).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of minutes (seconds / seconds in a minute = seconds / 60 - integer division).

time.secondsToNanos

<Integral T> Int64 time.secondsToNanos(T seconds)

Converts the time duration given in seconds to nanoseconds (as integer).

Parameters:

- seconds - the time duration given in seconds (non-nullable).

Returns:

the time duration expressed in terms of nanoseconds (seconds x (nanos in a second) = seconds x (1000 x 1000 x 1000)).

time.utcDateTimeFromDate

List(Int32) time.utcDateTimeFromDate(Int32 year, Int32 month, Int32 day)

Produces the UTC date-time components ([year, month, mday, hour, minute, second, wday, yday, isdst]) according to the given date information (year, month and day)

year: Year as a decimal number

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

Parameters:

- year - year (in the range [1:9999]) (non-nullable).
- month - month (in the range [1:12]) (non-nullable).
- day - the day of the month (in the range [1:31]) (non-nullable).

Returns:

the UTC date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst].

time.utcDateTimeFromDateTime

List(Int32) time.utcDateTimeFromDateTime(Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second)

Produces the UTC date-time components ([year, month, mday, hour, minute, second, wday, yday, isdst]) according to the given date and time information (year, month, day, hour, minute and second)

year: Year as a decimal number

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

Parameters:

- year - the year (in the range [1:9999]) (non-nullable).
- month - the month within the year (in the range [1:12]) (non-nullable).
- day - the day within the month (in the range [1:31]) (non-nullable).
- hour - the hour within the day (in the range [0:23]) (non-nullable).
- minute - the minute within the hour (in the range [0:59]) (non-nullable).
- second - the second within the minute (in the range [0:59]) (non-nullable).

Returns:

the UTC date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst].

time.utcTime

List(Int32) time.utcTime(Double value)

Converts the time in seconds since the Epoch to the UTC date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst]

year: Year as a decimal number

month: Month, in the range [1:12]

mday: Day of the month, in the range [1:31]

hour: Hours, in the range [0:23]

minute: Minutes, in the range [0:59]

second: Seconds, in the range [0:59]

wday: Day of the week, in the range [0:6], Monday is 0

yday: Day of the year, in the range [1:366]

isdst: 1 if daylight saving time is in effect, 0 otherwise.

Parameters:

- value - the time (fractional seconds since the Epoch) (non-nullable).

Returns:

the UTC date-time components in the format: [year, month, mday, hour, minute, second, wday, yday, isdst].

Generic Type Classes

Integral

- Int16
- Int32
- Int64

Numeric

- Double
- Int16
- Int32
- Int64

Primitive

- Bool
- Double
- Int16
- Int32
- Int64
- String

Temporal

- DateTime