Advanced Analytics

CAPSTONE PROJECT | ASSIGNMENT 5

In this portion of the capstone project we continue building our light bulb application and create Analytics that are triggered by data flow events.

To that end, in the following sections we:

1. Create simple Data Flow Events (DFEs).
2. Create compound Data Flow Events (DFEs).
3. Create and test Analytics.
4. Declare, implement, and perform a MapReduce Job.

As you may recall from the lectures, Analytics on the C3 IoT Platform are a powerful tool that enable you to respond automatically, with logic you've described, to data changes that are important to you.

As you might also recall, MapReduce is a widely-applied concept that facilitates the processing of large amounts of data efficiently, and this lesson shows you how to use MapReduce on the C3 IoT Platform.

With that in mind, let's get going!

1. CREATING ANALYTICS

For this project we write three Analytics, each of which creates a `SmartBulbEvent` that alerts us when a `SmartBulb` has:

- Been switched on for longer than some threshold time
- Overheated
- Failed

We guide you through the process of creating the first Analytic. After that, you are tasked with writing the DFEs and Analytics for the other two criteria on your own.

A. Creating Data Flow Events

In this section we create the simple and compound DFEs that we pass to our Analytic, which alerts us if a smart bulb surpasses a certain lifetime threshold.

Smart Bulb Long Life

Our simple DFE `SmartBulbLongLife` is the input to our compound DFE, `SmartBulbLongLifeInput`, and provides a link to the source type, `SmartBulb`, whose data we are interested in.

```
@DFE(period='Hour', interval='Hour', metric='DurationOnInHours',
flattenWindows:true)
type SmartBulbLongLife mixes TSDataFlowEvent<SmartBulb>
```

As you can see we specify parameters for our DFE using the `@DFE` annotation. Save this as `SmartBulbLongLife.c3typ` in the `/src` folder.

Smart Bulb Long Life Input

`SmartBulbLongLifeInput` is our compound DFE which is the input to our Analytic, `SmartBulbLongLifeAlert`. The compound DFE fires the Analytic when it's triggered by a data event.

```
@DFE(period='Hour', interval='Hour')
type SmartBulbLongLifeInput mixes CompoundDataFlowEvent<SmartBulb> {
  bulbLife: SmartBulbLongLife
}
```

Again, we specify parameters for our DFE using the `@DFE` annotation. Save this as `SmartBulbLongLifeInput.c3typ` in the `/src` folder.

B. Creating Analytics

In this section we show you how to create an Analytic. We continue our guided example with `SmartBulbLongLifeAlert`.

Like DFEs, Analytics are types and can therefore be created just like any other type.

Smart Bulb Long Life Alert

Our Analytic `SmartBulbLongLifeAlert` takes `SmartBulbLongLifeInput` as its input and outputs a `SmartBulbEvent` that stores our alert.

```
type SmartBulbLongLifeAlert mixes Analytic<SmartBulbLongLifeInput,
SmartBulbEvent> {
  process : ~ js server
}
```

Save this as `SmartBulbLongLifeAlert.c3typ` in the `/src` folder. We need to override the `process()` function, which defines our response to a data event.

C. Implementing Analytics

The final step will be less guided than the previous steps.

We need to write the implementation of the `process()` function, which will define our response to a data event. Use the implementation of the `SmartBulbOverheatAlert` Analytic example from lecture, which is shown below, as a guide. The code can be found in the provided `SmartBulbOverheatAlert.js` file.

The only things that should be changed from this example for `SmartBulbLongLifeAlert` are:

- the `string` inside the `C3.logger()` command.
- The value and name of the threshold variable, which should be set to 10500 hours.
- The `eventCode`.
- The field accessed from `input` for `data` and `dates` (`bulbLife` – see `SmartBulbLongLifeInput`).

All else will remain the same.

Save your implementation in the same folder as your `SmartBulbLongLifeAlert.c3typ`. You should call this file `SmartBulbLongLifeAlert.js`.

When you are finished, be sure to provision the application.

```
var log = C3.logger("SmartBulbOverheatAlert");

var TEMP_THRESHOLD = 95;

function process(input) {
  var data = input.temperature.data(),
      dates = input.temperature.dates();
  for (var i = 0; i < data.length; i++) {
    // If the temperature is greater than the threshold then we need to update the source object
    if (data.at(i) > TEMP_THRESHOLD) {
      return SmartBulbEvent.make({
        smartBulb: input.source,
        eventCode: "OVERHEAT",
        eventType: "HEALTH",
        start: dates.at(i),
        end: dates.at(i+1)
      });
    }
  }
}
```

Input the time series object returned from the DFE

Implementation of the `process()` function on the `SmartBulbOverheatAlert` Analytic. Consult the "Defining Analytics" lecture for additional information.

D. Testing Analytics

Once you have finished writing the DFEs and Analytics, it is time to test them. Before we can test, however, we need to change a few of our configurations to that we can fire Analytics for a period far in the past. Launch the console and type the following commands:

1. `myvariable = TenantConfig.get('ACE-StartOfTime')`
2. `myvariable.value = -3650`
3. `TenantConfig.upsert(myvariable)`

With these commands, we enabled the Analytics Container Engine to run Analytics on data from up to 10 years ago.

The `AnalyticsContainer` type has a function `fireAnalytics`, which allows us to test our Analytics. Enter the following command into the console:

```
 AnalyticsContainer.fireAnalytics([{typeId:
Tag.getTypeId('lightbulb','SmartBulb'),
   objId: 'SMBLB1', timeRanges: [{start: "2011-01-01", end: "2016-01-01"}]}],
   ['SmartBulbLongLifeAlert'], {forceReEval: true})
```

As you can see, we are firing our `SmartBulbLongLifeAlert` Analytic on 5 years of measurement data from the `SmartBulb` object with `id = "SMBLB1"`.

If we've written our DFEs and Analytic correctly, running this command should write alerts by generating `SmartBulbEvent` objects. Run `c3Grid(SmartBulbEvent.fetch())` in the console to see what alerts the Analytic produced.

E. Two More Analytics

Now that you have a better understanding of the how to create DFEs and Analytics, you are tasked with creating two more Analytics, one that generates an alert if a smart bulb is overheating, and another than generates an alert if a smart bulb has failed.

The simple DFEs, compound DFEs, and Analytics will be declared and implemented very similarly to the `SmartBulbLongLifeAlert` example. These specifications indicate the differences:

1. Overheat Alert – generates a `SmartBulbEvent` if the smart bulb is overheating
    a. Simple DFE: `SmartBulbOverheat`
        i. `metric: AverageTemperature`
        ii. `flattenWindows: false`
    b. Compound DFE: `SmartBulbOverheatInput`
        i. `temperature: SmartBulbOverheat`
            1. This is an advanced reference field
    c. Analytic: `SmartBulbOverheatAlert`
        i. Consult the "Defining Analytics" lecture

2. Failure Alert – generates a `SmartBulbEvent` if the smart bulb has failed
    a. Simple DFE: `SmartBulbFailure`
        i. `metric: HasEverFailed`
    b. Compound DFE: `SmartBulbFailureInput`
        i. `hasFailed: SmartBulbFailure`
            1. This is an advanced reference field
    c. Analytic: `SmartBulbFailureAlert`
        i. `process()` function implemented similarly to the previous two Analytics Note that we are checking whether the bulb has failed (1) or not (0), not comparing to a threshold as we did previously.

When you finish creating the DFEs and Analytics, provision the application and test the Analytics using the `fireAnalytics` function as we did previously.

Now, we switch gears and turn our focus to MapReduce and how it is used on the C3 IoT platform.

2. MAP REDUCE JOBS

In this exercise we want to find the total power expended by smart bulbs for each manufacturer over a certain time. To this end, we will do two things:

1. Configure and implement a MapReduce job.
2. Create a MapReduce instance.
3. Run a MapReduce job.

Before we can configure a MapReduce job, however, we need to create supporting types that will store aggregate power consumption data per manufacturer.

Aggregate Consumption By Manufacturer

The `AggregateConsumptionByManufactuer` type stores the aggregate power consumption data per manufacturer.

```
/**
 * A single aggregate power measurement for a manufacturer
 */
@db(compactType=true,
  datastore='cassandra',
  partitionKeyField='parent',
  persistenceOrder='start',
  persistDuplicates=false,
  shortId=true,
  shortIdReservationRange=100000)
entity type AggregateConsumptionByManufacturer schema name "AGG_CNS_MNFCT" {
  parent: Manufacturer
  start: datetime
  end: datetime
  aggregateConsumption: double
}
```

Save this as `AggregateConsumptionByManufactuer.c3typ` in the `/src` folder.

A. Creating MapReduce Job

MapReduce jobs are entity types that extend `MapReduce`. To run a MapReduce job on the C3 IoT platform, you must create a type that establishes the argument types and relevant fields that will be needed for the job. We also declare the `map()` and `reduce()` functions on this type.

Create a type for our MapReduce called `AggregateConsumptionByManufacturerJob`.

```
/**
 * Map reduce job that returns the aggregate power consumption per
manufacturer
 */
entity type AggregateConsumptionByManufacturerJob extends
MapReduce<SmartBulb, string, double, double> type key 'AGG_CNS_MNFCT' {

  // Start of the evaluation interval
  startDate: datetime

  // End of the evaluation interval
  endDate: datetime

  // Metric evaluation interval
  interval: string

  // Map function
  map:     ~ js server

  // Reduce function
  reduce:  ~ js server

}
```

Save this as `AggregateConsumptionByManufacturerJob.c3typ` in the `/src` folder.

B. Adding MapReduce Implementation

Now that we have created the `AggregateConsumptionByManufacturerJob` type for our MapReduce job, we need to add the MapReduce implementation. We have provided you the implementations for both `map()` and `reduce()` in the `AggregateConsumptionByManufacturerJob.js` file that you can put in the same folder as `AggregateConsumptionByManufactuerJob.c3typ`.

Provision the application when you are finished.

C. Running MapReduce Job

We have created the supporting types as well as the MapReduce job and its implementation. It is time to run the job and see the results! Make sure to provision your new files, and then create an instance of `AggregateConsumptionByManufacturerJob` in the console:

1. ```
   job = AggregateConsumptionByManufacturerJob.make({batchSize: 10, limit: -1,
   include: "id, manufacturer", startDate: "2011-01-01", endDate: "2011-02-01",
   interval: "MONTH", id: "agg_job"});
   ```

2. ```
   job = AggregateConsumptionByManufacturerJob.create(job);
   ```

Then, start the job:

- ```
  AggregateConsumptionByManufacturerJob.start(job);
  ```

You can check the job execution status by running the following command:

- ```
  AggregateConsumptionByManufacturerJob.status(job)
  ```

After the status shows “completed”, find your results by running a `fetch()` command on your `AggregateConsumptionByManufacturer` type:

- ```
  c3Grid(AggregateConsumptionByManufacturer.fetch())
  ```

Congratulations! In the next section you’ll learn how to use machine learning in your applications.
  ```