# Methods & Metrics

**CAPSTONE PROJECT | ASSIGNMENT 4**

This portion of the capstone project is focused on writing methods and metrics.

To this end, the tasks are twofold:

1. Write methods for the `SmartBulb` type.
2. Write simple and compound metrics for the `SmartBulb` type.

The purpose of these tasks is to give you a better understanding of what methods and metrics are, and how they are written.

By the end of this portion of the project we will have written methods and metrics that can produce relevant and helpful insights from our data, insights that will provide the basis for later portions of this project.

## 1. FUNCTIONS

We start by writing methods for the `SmartBulb` type that will compute:

- The life span of a particular smart bulb.
- The average life span of all our smart bulbs.
- The smart bulb with the longest life span.
- The smart bulb with the shortest life span.

The following sections show you how to declare and implement methods.

### A. Declaring Methods

The first step to writing a method is to declare it for a particular type. We declare all the methods in this portion of the project for the `SmartBulb` type. Add the following line of code to your `SmartBulb.c3typ` file to declare the `lifeSpanInYears()` method.

```
/**
 * Returns the life span of this smartBulb
 */
lifeSpanInYears: function(bulbId: string): double js server
```

We have now declared `lifeSpanInYears()` on our `SmartBulb` type.

### B. Implementing Methods

Now that we have defined the method on the `SmartBulb` type, we can implement it using JavaScript. We have provided you with a `SmartBulb.js` file for that purpose. Save this file in the same folder as your `SmartBulb.c3typ` file. Next time you provision you will be able to call this method.

### C. Three More Methods

Now that you know how to declare and implement methods, you are tasked with implementing three more methods specified below:

1. Longest Life Span
    a. `Method Name: longestLifeSpanBulb`
    b. `Returns: SmartBulb`
    c. `Description:` Return the `SmartBulb` object with the longest life span.

2. Shortest Life Span
    a. `Method Name: shortestLifeSpanBulb`
    b. `Returns: SmartBulb`
    c. `Description:` Return the `SmartBulb` object with the shortest life span.

3. Average Life Span
    a. `Method Name: averageLifeSpan`
    b. `Returns: double`
    c. `Description:` Return a `double` representing the average life span (in years) across all `SmartBulb` objects.

Declare and implement these methods in the `SmartBulb.js` file, as you did for `lifeSpanInYears()`. As you begin writing, you may find the following suggestions helpful:

- Use `lifeSpanInYears()` in your implementation for each of the three methods

- Understanding `SmartBulb.fetch()` will be useful. These console commands will help:

- `c3ShowType(FetchSpec)`
- `c3ShowType(FetchResult)`

Good luck! When you have finished, provision and then check that your methods work as intended by provisioning and then running them in the console.

## 2. METRICS

Methods allow us to fetch a specific value, but don't provide a scalable way to process big time series data. This is why we use metrics. The C3 IoT metrics engine gives you a powerful tool to analyze large amounts of data coming at rapid intervals.

In this section we write simple metrics that will allow us to produce time series of several features of our smart bulbs, such as:

- Average voltage
- Average power
- Average lumens
- Average temperature
- Status

Using these simple metrics, we write compound metrics that produce time series which tell us:

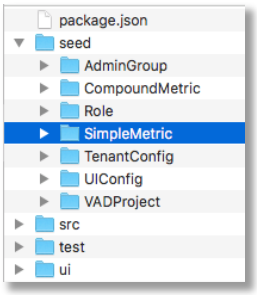
- How long a smart bulb has been on.
- If a smart bulb is defective.
- If a smart bulb has ever failed.
- The number of times a smart bulb has been switched on or off.
- A risk score of whether a smart bulb will fail next month.

In the following sections we create and evaluate standard simple metrics, time series declaration metrics, and compound metrics.

### A. Simple Metrics

Simple metrics are instructions for how to produce a time series from the data on a certain source type. A simple metric specifies the type it is getting data from, the field on that type that points to the time series data it's trying to access, and an expression to evaluate on that data.

Metrics are json files that reside in the `/seed` folder, which is on the same level as your `/src` folder. Within this folder, create separate folders called `/SimpleMetric` and `/CompoundMetric` for simple and compound metrics respectively (they must have these exact names in order for your package to provision correctly):




We'll start by writing a metric that will return an average of a smart bulb's voltage measurement data:

```
{
  "id": "AverageVoltage_SmartBulb",
  "name": "AverageVoltage",
  "description": "Average voltage over time of the SmartBulb",
  "path": "bulbMeasurements",
  "expression": "avg(avg(normalized.data.voltage))",
  "srcType": "SmartBulb"
}
```

You can save this metric as `AverageVoltage_SmartBulb.json` in the `/SimpleMetric` folder.

You have created your first simple metric! Using the format above, create four more simple metrics and save them in the `/seed` folder. These metric definitions will be identical to the above, except for the following values (you will need to replace the ??? with the appropriate value):

1. Average Power
    a. `id: ???_SmartBulb`
    b. `name: AveragePower`
    c. `description: "Average power over time of the SmartBulb"`
    d. `expression: "avg(avg(normalized.data.???))"`
    e. Save as `AveragePower_SmartBulb.json`
2. Average Lumens
    a. `id: AverageLumens_SmartBulb`
    b. `name: ???`

   c. `description: "Average lumens over time of the SmartBulb"`
   d. `expression: "avg(avg(normalized.data.???))"`
   e. Save as `AverageLumens_SmartBulb.json`
3. Average Temperature
   a. `id: AverageTemperature_???`
   b. `name: AverageTemperature`
   c. `description: "Average temperature over time of the SmartBulb"`
   d. `expression: "avg(avg(normalized.data.???))"`
   e. Save as `AverageTemperature_SmartBulb.json`
4. Status
   a. `id: ???_???`
   b. `name: Status`
   c. `description: "Status over time of the SmartBulb"`
   d. `expression: "???(???(normalized.data.???))"`
   e. Save as `Status_SmartBulb.json`

**Hint:** For the Status metric, what expression engine function should we use for aggregation across time and aggregation across space? Why does `avg` not make sense if the value for status is only 1 or 0? What function makes sense if we want the `max` value?

### B. Time Series Declaration Metrics

Not all timed data will have a stable interval. For example, the status (on or off) of a smart bulb can change at any time, unlike a measurement such as temperature which we receive at regular intervals. To evaluate this kind of data, we use a time series declaration metric.

We will do this for a metric named `PowerGridStatus`, which will tell us the status of the power grid supplying a `SmartBulb` type.

```
{
  "id": "PowerGridStatus_SmartBulb",
  "name": "PowerGridStatus",
  "description": "Status (1 or 0) indicating ON or OFF of the PowerGrid over
time",
  "srcType": "SmartBulb",
  "path": "fixtureHistory.to.apartment.building",
  "tsDecl": {
    "data": "gridStatusSet",
    "value": "value",
    "treatment": "PREVIOUS",
    "start": "timestamp"
  }
}
```

Save this file as `PowerGridStatus_SmartBulb.json` in the `/SimpleMetric` folder.

Now, we will do this for a metric named `PowerGridStatus`, which will tell us the status of the power grid supplying a `Building` type.

```
{
  "id": "PowerGridStatus_Building",
  "name": "PowerGridStatus",
  "description": "Status (1 or 0) indicating ON or OFF of the PowerGrid over
time",
  "srcType" : "Building",
  "tsDecl": {
    "data": "gridStatusSet",
    "value": "value",
    "treatment": "PREVIOUS",
    "start": "timestamp"
  }
}
```

Save this file as `PowerGridStatus_Building.json` in the `/SimpleMetric` folder.

Now, you have declared the power grid status measurements as time series and can evaluate them as such!

### C. Compound Metrics

Now that we have created our simple metrics, we can combine them or add additional functionality with compound metrics. We will use these compound metrics later in the project in our Analytics and machine learning models.

For example, we will be interested in having a time series that will show the total duration that a smart bulb has been on, or the number of times that a smart bulb has been switched on and off.

#### Is Defective

The `IsDefective` metric tells us whether a smart bulb is defective or not.

```
{
    "name": "IsDefective",
    "id": "IsDefective",
    "description": "Indicator (1 or 0) if the smart bulb died",
    "expression": "sum(eval('HOUR', AverageLumens == 0 && Status == 1)) ? 1 :
0"
}
```

Save this file as `IsDefective.json` in the `/CompoundMetric` folder.

As you can see from the `expression` field, we are looking for situations when a smart bulb was on (`Status` is 1) while at the same time it was not emitting any light (average lumens is 0). If both these requirements are met, the metric will produce a 1, which tells us that the bulb is defective. In all other cases we will get a 0.

#### Duration On In Hours

The `DurationOnInHours` metric tells us how long a smart bulb has been on.

```
{
    "name": "DurationOnInHours",
    "id": "DurationOnInHours",
    "description": " total amount of time a smart bulb is switched on up to
the interval",
    "expression": "rolling('SUM',sum(eval('HOUR', Status)))"
}
```

Save this file as `DurationOnInHours.json` in the `/CompoundMetric` folder.

As you can see from the `expression` field, we are taking a rolling sum on the time series produced by the `Status` simple metric. We are evaluating at every hour, which gives us the total number of hours that a smart bulb has been on.

**Switch Count**

We hypothesize that the number of times that a smart bulb has been switched on or off over a certain period of time will be predictive of its failure. For that purpose, let us create a metric that allows us to extract that insight.

```
{
    "name": "SwitchCount",
    "id": "SwitchCount",
    "description": "The number of times a smart bulb is switched on or off
beginning at the 'start' given",
    "expression": "sum(eval('HOUR', abs(rollingDiff(Status))))"
}
```

Save this file as `SwitchCount.json` in the `/CompoundMetric` folder.

As you can see from the `expression` field, we are using the `Status` simple metric to find the number of times a smart bulb is switched on or off beginning at the time in the `startDate` field.

We can take this metric a step further by creating another compound metric `SwitchCountPreviousWeek` that uses `SwitchCount` to find the switch count for the previous week.

```
{
    "name": "SwitchCountPreviousWeek",
    "id": "SwitchCountPreviousWeek",
    "description": "The total number of times a smart bulb is switched on or
off up in the previous week of the interval",
    "expression": "window('SUM', SwitchCount, -7, 7)"
}
```

Save this file as `SwtichCountPreviousWeek.json` in the `/CompoundMetric` folder.

As you can see from the `expression` field, we have specified a range for which this metric will run as a unit of 7. In later portions of the project when we use `SwitchCountPreviousWeek` for our ML model, we will evaluate it on the DAY in order to produce the desired week range we are looking for.

### Has Ever Failed

To help us later in the project, we want to have the ability to filter out a training set for our ML algorithms by selecting smart bulbs that have never failed. The solution is the `HasEverFailed` metric, which tells us which smart bulbs have failed.

```
{
    "name": "HasEverFailed",
    "id": "HasEverFailed",
    "description": "Indicates (1 or 0) if the SmartBulb has failed during
evaluated interval",
    "expression": "rolling('SUM',IsDefective) ? 1 : 0"
}
```

Save this file as `HasEverFailed.json` in the `/CompoundMetric` folder.

### Will Fail Next Month

We now have all the metrics defined that we will use as the training features in our future ML model. However, we are missing a metric that we can use as the training label for our ML model. For this purpose, we define compound metric `WillFailNextMonth`, which tells us if a smart bulb will fail in the next month.

```
{
    "name": "WillFailNextMonth",
    "id": "WillFailNextMonth",
    "description" : "Indicator (1 or 0) if the smart bulb died within the
next 30 days",
    "expression": "window('MAX', IsDefective, 0, 30) > 0"
}
```

Save this file as `WillFailNextMonth.json` in the `/CompoundMetric` folder.

Provision your application now.

We now have all our metrics and can start exploring some of the insights they give us. In the next section we evaluate the metrics we just created.

### D. Evaluating Metrics

To evaluate metrics on a type, the type must mix in `MetricEvaluatable`. We have defined our `SmartBulb` to be `MetricEvaluatable`, so we can go straight ahead and evaluate metrics for this type.

- Run `c3ShowType(MetricEvaluatable)` to see all functions that `MetricEvaluatable` provides.
    - `evalMetric()` is the basic metric evaluation function, it allows the evaluation of one metric on one instance of the source type
    - `evalMetrics()` allows you to evaluate multiple metrics or multiple instances of types
- `c3Viz()` is a very useful console function to visualize output from `evalMetrics()` results.

Launch the console. The command below will create a variable `TS` which stores the `EvalMetricsResult` from `evalMetrics()`, which evaluates the `AverageVoltage` metric on `SmartBulb` objects with id's "SMBLB1" and "SMBLB2" over the full year of 2012 at an hourly interval.

```
TS = SmartBulb.evalMetrics({ids: ["SMBLB1", "SMBLB2"],expressions:
["AverageVoltage"], start: "2012-01-01T00:00:00Z", end: "2013-01-
01T00:00:00Z", interval: "HOUR" });
```

Run the `c3Viz()` command on `TS` to visualize the metric:

```
c3Viz(TS)
```

Now you should have a better understanding of how to create and evaluate metrics. This will be a very important part of developing on the C3 Platform as metrics are used both in UI visualization as well as inputs for the ML models we will be building later on in the course!