Data Integration & Time Series

CAPSTONE PROJECT | ASSIGNMENT 3

This portion of the capstone project is focused on data integration and time series.

To this end, the tasks are twofold:

1. Create canonicals and canonical transforms.
2. Load data into the C3 IoT Platform using the command line.

Let's get started!

1. CLEAN UP

Run the following commands in the console to clean out the temporary data you created in the last module.

```
SmartBulb.removeAll();

Fixture.removeAll();

SmartBulbToFixtureRelation.removeAll();
```

2. MEASUREMENTS

In this section, we are integrating measurement data from our smart bulbs into our data model. To do this, we must first create types that can store this measurement data. The data to be stored includes:

- Lumens
- Power
- Temperature
- Voltage
- Status

To represent this data as a time series we must create two types, one mixing `TimeseriesDataPoint<P>` and the other mixing `TimeseriesHeader<TDP>`.

The type mixing `TimeseriesDataPoint<P>` stores measurement data. We call this type `SmartBulbMeasurement`. The type mixing `TimeseriesHeader<TDP>` serves as a header for the measurement data, organizing it and specifying how it should be normalized. We call this type `SmartBulbMeasurementSeries`.

Smart Bulb Measurement

As previously stated, the `SmartBulbMeasurement` type stores measurement data from smart bulbs, and mixes in `TimeseriesDataPoint<P>`.

We want to define how we treat our measurement data during normalization by adding a treatment option to each field, as you'll see below. Treatment options are basically `strings` that define how we want aggregate time series values. To learn about what treatment options are available, it's recommended you run `c3ShowType(Treatment)` in the console.

Note that we will not be able to mix in `TimeseriesDataPoint<P>` because we haven't created `SmartBulbMeasurementSeries` yet. We will fix this after creating `SmartBulbMeasurementSeries`.

```
/**
 * A single measurement taken from a single {@link SmartBulb}.
 */
entity type SmartBulbMeasurement schema name 'SMRT_BLB_MSRMNT' {

  // The measured number of lumens.
  @ts(treatment='rate')
  lumens: double

  // The measured power consumption.
  @ts(treatment='rate')
  power: double

  // The measured temperature.
  @ts(treatment='rate')
  temperature: double

  // The measured voltage.
  @ts(treatment='rate')
  voltage: double

  // The status of the smart bulb (on or off).
  @ts(treatment='previous')
  status: int

}
```

Because we will be uploading a large data set, we want this type to be stored in Cassandra. This will ensure that our application performs optimally. Add this annotation to the top of your `.c3typ` file:

```
/**
 * A single measurement taken from a single {@link SmartBulb}.
 */
@db(datastore='cassandra',
    partitionKeyField='parent',
    persistenceOrder='start',
    persistDuplicates=false,
    compactType=true,
    shortIdReservationRange=100000)
entity type SmartBulbMeasurement schema name 'SMRT_BLB_MSRMNT'{
 ...
}
```

Save this file as `SmartBulbMeasurement.c3typ` in the `/src` folder.

Smart Bulb Measurement Series

The `SmartBulbMeasurementSeries` type is a header for our time series. This type mixes in `TimeseriesHeader<TDP>`. Some important fields in `TimeseriesHeader<TDP>` are shown here:

```
data: [TDP]      // Reference to TimeseriesDataPoints
   (ex:SmartBulbMeasurement)

/* Several fields on how to treat data */
grain: string             // Grain at which data should be normalized
interval: string          // Interval at which data should be normalized
duplicateHandling: string // Specify how normalization should handle
duplicates
overlapHandling: string   // Specify how normalization should handle overlap
interpolator: string      // Specify interpolation mechanism to apply to gaps
  ...
```

These fields refer to a collection of data points for a particular smart bulb and specify several features about how we normalize our data.

Let's create the `SmartBulbMeasurementSeries` type. It has two fields: a reference to the smart bulb that this time series data came from, and a `string` specifying how to treat this data.

```
/*
 * A series of measurements taken from a single {@link SmartBulb}.
 */
entity type SmartBulbMeasurementSeries mixes
TimeseriesHeader<SmartBulbMeasurement> schema name "SMRT_BLB_MSRMNT_SRS" {

  // The {@link SmartBulb} for which measurements were taken.
  smartBulb: SmartBulb

  // The aggregation/disaggregation {@link Treatment} to use for the
measurements.
  treatment: string enum('previous', 'rate', 'integral')

}
```

Save this file as `SmartBulbMeasurementSeries.c3typ` in the `/src` folder.

Smart Bulb Measurement

Revisit the `SmartBulbMeasurement` type so that we can mix in `TimeseriesDataPoint<P>`. The `TimeseriesDataPoint<P>` type provides a few important fields:

```
parent: P        // Reference to a TimeseriesHeader
(ex:SmartBulbMeasurementSeries)
start: datetime  // Beginning of the measurement
end: datetime    // End of the measurement (only for Interval Timeseries)
```

These fields enable related measurement data to be organized under the same header and keep track of the time that the measurement data was collected.

Mix in `TimeseriesDataPoint<P>` by adding the bolded phrase to your type declaration:

```
entity type SmartBulbMeasurement mixes
TimeseriesDataPoint<SmartBulbMeasurementSeries> schema name 'SMRT_BLB_MSRMNT'
{
 ...
}
```

Smart Bulb

We want to be able to look at the measurement data for a particular smart bulb. To that end, we add a collection field to the `SmartBulb` type so that each `SmartBulb` will have a historical collection of time series with its measurement data.

```
entity type SmartBulb extends LightBulb mixes MetricEvaluatable type key
'SMRT_BLB' {
   ...

   // This bulb's historical measurements.
   bulbMeasurements: [SmartBulbMeasurementSeries](smartBulb)

   ...

}
```

3. CANONICALS AND TRANSFORMS

We have created types that can store and organize our measurement data. The next step is to create the canonicals and canonical transforms that will allow us to load external data into our data model.

As a reminder, canonicals are types that serve as an interface between external data and the C3 Platform. Any external data coming onto the Platform is first loaded into canonicals before it is passed into entity types.

Transforms are the types that determine how data is passed from canonicals to entity types. They define a mapping between the fields on a canonical to the fields on an entity type.

Canonicals and canonical transforms, like entity types, should be saved in the `/src` folder with the `.c3typ` extension.

A. Making Canonicals

Our data will be uploaded to our canonicals from .csv files. Therefore, the fields on the canonical should be associated with the column headers in the .csv that the canonical will be receiving its data from. To understand why certain fields are included in our canonicals, look at the column headers in the associated .csv files:

- `CanonicalSmartBulb : SmartBulb.csv`
- `CanonicalFixture : Fixture.csv`
- `CanonicalSmartBulbToFixtureRelation : SmartBulbFixture.csv`
- `CanonicalPowerGridStatus : PowerGridStatus.csv`
- `CanonicalSmartBulbMeasurement : SmartBulbMeasurement.csv`

Without further ado, let's make some canonicals!

Canonical Smart Bulb

The `CanonicalSmartBulb` type accepts static data to map to our `SmartBulb`.

```
/**
 * This type represents the raw data that will represent {@link
SmartBulb}information.
 */
type CanonicalSmartBulb mixes Canonical<CanonicalSmartBulb> {

  // This represents the manufacturer of a {@link LightBulb}
  Manufacturer: string

  // This represents the bulb type of a {@link LightBulb}
  BulbType:     string

  // This represents the wattage of a {@link LightBulb}
  Wattage:     decimal

  // This represents the id of a {@link LightBulb}
  SN:           string

  // This represents the start date of a {@link LightBulb}
  StartDate:    datetime

  // This represents the latitude of a {@link SmartBulb}
  Latitude:     double

  // This represents the longitude of a {@link SmartBulb}
  Longitude:    double

}
```

You have created your first canonical, congratulations! We now need to repeat this process for the other types that we want to load data on. Follow the instructions below to create those canonicals!

Canonical Smart Bulb Measurement

The `CanonicalSmartBulbMeasurement` type accepts timed measurement data to map to our `SmartBulbMeasurement` and `SmartBulbMeasurementSeries`.

```
/**
 * This type represents the raw data that will represent {@link
SmartBulbMeasurement} information.
 */
type CanonicalSmartBulbMeasurement mixes
Canonical<CanonicalSmartBulbMeasurement> {

  // This represents the datetime to use for the start of {@link
SmartBulbMeasurement}
  TS:       datetime

  // This represents the id of the {@link SmartBulb}
  SN:       string

  // This represents the status measurement
  Status:   integer

  // This represents the watts measurement
  Watts:    double

  // This represents the lumens measurement
  Lumens:   double

  // This represents the voltage measurement
  Voltage:  double

  // This represents the temperature measurement
  Temp:     double

}
```

Canonical Fixture

`CanonicalFixture` accepts data to map to the `Fixture`, `Apartment`, and `Building` type.

```
/**
 * This type represents the raw data that will represent {@link Fixture}
information.
 */
type CanonicalFixture mixes Canonical<CanonicalFixture> {

  // This represents the id of a {@link Fixture}
  fixture:    string

  // This represents the id of an {@link Apartment}
  apartment:  string

  // This represents the id of a {@link Building}
  building:   string

}
```

Canonical Smart Bulb To Fixture Relation

`CanonicalSmartBulbToFixtureRelation` accepts data to map to our `SmartBulbToFixtureRelation` type.

```
/**
 * This type represents the raw data that will represent {@link
SmartBulbToFixtureRelation} information.
 */
type CanonicalSmartBulbToFixtureRelation mixes
Canonical<CanonicalSmartBulbToFixtureRelation> {

  // This represents the id of a {@link SmartBulb}
  SN:       string

  // This represents the id of a {@link Fixture}
  fixture:  string

  // The date time to use for the start of a
{@linkSmartBulbToFixtureRelation}
  start:    datetime

  // The date time to use for the end of a {@link SmartBulbToFixtureRelation}
  end:      datetime

}
```

Canonical Power Grid Status

`CanonicalPowerGridStatus` accepts data to map to our `Building` and `Manufacturer` type.

```
/**
 * This type represents the raw data that will represent {@link
PowerGridStatus} information.
 */
type CanonicalPowerGridStatus mixes Canonical<CanonicalPowerGridStatus> {

  // This represents the datetime to use for the start of {@link
PowerGridStatus}
  TS:       datetime

  // This represents the id of the {@link Building}
  Building: string

  // This represents the status ON or OFF (1 or 0) for the power grid of a
{@Building}
  Status:   int

}
```

Canonical Smart Bulb Event

`CanonicalSmartBulbEvent` accepts data to map to our `SmartBulbEvent` type.

```
/**
 * This type represents the raw data that will represent {@link
SmartBulbEvent} information.
 */
type CanonicalSmartBulbEvent mixes Canonical<CanonicalSmartBulbEvent> {

  // This represents the id for a {@link SmartBulbEvent}
  id:         string

  // This represents the id of a {@link SmartBulb}
  bulb:       string

  // This represents the eventCode of a {@link SmartBulbEvent}
  eventCode:  string

  // This represents the eventType of a {@link SmartBulbEvent}
  eventType:  string

  // This represents a datetime to use as the start of a {@link
SmartBulbEvent}
  start:      datetime

  // This represents a datetime to use as the end of a {@link SmartBulbEvent}
  end:        datetime

}
```

B. Configuring File Source Systems

A source system is a system that will be sending data to C3 to be processed, such as an S3 bucket or CSV file.

We need to configure where the resulting files (CSV, JSON, Parquet, Avro, etc.) will be stored in the file system to be associated with the Types.

If this isn't specified, a default File Source System will be created. We will create our own. We need one for each Canonical we have created. Each of these files is a `.json` file.

Create a folder called `FileSourceCollection` in your `/seed` folder. All of these files will be saved there.

Here is an example for `CanonicalSmartBulb`.

CanonicalSmartBulb.json

```
{
  "id": "CanonicalSmartBulb",
  "name": "CanonicalSmartBulb",
  "source": {
    "typeName": "CanonicalSmartBulb"
  },
  "sourceSystem": {
        "id": "Legacy"
  }
}
```

Now create these files for the other canonicals.

C. Making Transforms

Now that we have created canonicals and configured the file source systems, the next step is to create canonical transforms. This allows us to map our canonical types to the entity types in our data model such as `SmartBulb`, `Fixture`, `SmartBulbMeasurement`, and so forth.

Remember, each canonical transform maps one canonical to one entity type. You will be mapping the data in the canonical types to entity types with operations expressed in the canonical transform's fields.

TransformCanonicalSmartBulbToSmartBulb.c3typ

`TransformCanonicalSmartBulbToSmartBulb` maps fields in `CanonicalSmartBulb` to fields in `SmartBulb`.

```
/**
 * This type encapsulates the data flow from the {@link CanonicalSmartBulb}
to the {@link SmartBulb} type.
 */
type TransformCanonicalSmartBulbToSmartBulb mixes SmartBulb transforms
CanonicalSmartBulb {

  id:              ~ expression "SN"
  manufacturer:    ~ expression {id: "Manufacturer"}
  bulbType:        ~ expression "BulbType"
  wattage:         ~ expression "Wattage"
  startDate:       ~ expression "StartDate"
  latitude:        ~ expression "Latitude"
  longitude:       ~ expression "Longitude"
  lumensUOM:       ~ expression "{ \"id\": \"'lumen'\" }"
  powerUOM:        ~ expression "{ \"id\": \"'watt'\" }"
  temperatureUOM:  ~ expression "{ \"id\": \"'degrees_fahrenheit'\" }"
  voltageUOM:      ~ expression "{ \"id\": \"'volt'\" }"

}
```

TransformCanonicalSmartBulbMeasurementToSmartBulbMeasurement.c3typ

`TransformCanonicalSmartBulbMeasurementToSmartBulbMeasurement` maps fields in `CanonicalSmartBulbMeasurement` to fields in `SmartBulbMeasurement`.

```
/**
 * This type encapsulates the data flow from the {@link
CanonicalSmartBulbMeasurement} to the {@link SmartBulbMeasurement} type.
 */
type TransformCanonicalSmartBulbMeasurementToSmartBulbMeasurement mixes
SmartBulbMeasurement transforms CanonicalSmartBulbMeasurement {

  /**
   * Use the serial number field ("SN") of the canonical to create a
properly-formatted pointer to the parent {@link SmartBulbMeasurementSeries}
object.
   */
  parent:       ~ expression { id: "concat('SBMS_serialNo_',SN)" }
  lumens:       ~ expression "Lumens"
  power:        ~ expression "Watts"
  voltage:      ~ expression "Voltage"
  temperature:  ~ expression "Temp"
  start:        ~ expression "TS"
  end:          ~ expression "TS + period(1, 'HOUR')"
  status:       ~ expression "Status"

}
```

Transform Canonical Smart Bulb To Smart Bulb Measurement Series

`TransformCanonicalSmartBulbToSmartBulbMeasurementSeries` maps fields in `CanonicalSmartBulb` to fields in `SmartBulbMeasurementSeries`.

```
/**
 * This type encapsulates the data flow from the {@link
CanonicalSmartBulbMeasurement} to the {@link SmartBulbMeasurementSeries}
type.
 */
type TransformCanonicalSmartBulbMeasurementToSmartBulbMeasurementSeries mixes
SmartBulbMeasurementSeries transforms CanonicalSmartBulb {

  /**
   * The format of the id will be "SBMS_serialNo_serialNumber" to ensure that
no two series will be duplicates.
   */
  id:          ~ expression "concat('SBMS_serialNo_',SN)"
  smartBulb:   ~ expression { id: "SN" }
  interval:    ~ expression "'HOUR'"
  treatment:   ~ expression "'rate'"

}
```

Transform Canonical Fixture To Fixture

`TransformCanonicalFixtureToFixture` maps fields in `CanonicalFixture` to fields in `Fixture`.

```
/**
 * This type encapsulates the data flow from the {@link CanonicalFixture} to
the {@link Apartment} type.
 */
type TransformCanonicalFixtureToFixture mixes Fixture transforms
CanonicalFixture {

  id:          ~ expression "fixture"
  apartment:   ~ expression { id: "apartment" }

}
```

Transform Canonical Fixture To Apartment

`TransformCanonicalFixtureToApartment` maps fields in `CanonicalFixture` to fields in `Apartment`.

```
/**
 * This type encapsulates the data flow from the {@link CanonicalFixture} to
the {@link Apartment} type.
 */
type TransformCanonicalFixtureToApartment mixes Apartment transforms
CanonicalFixture {

  id:       ~ expression "apartment"
  building: ~ expression { id: "building" }

}
```

Transform Canonical Fixture To Building

`TransformCanonicalFixtureToBuilding` maps fields in `CanonicalFixture` to fields in `Building`.

```
/**
 * This type encapsulates the data flow from the {@link CanonicalFixture} to
the {@link Apartment} type.
 */
type TransformCanonicalFixtureToBuilding mixes Building transforms
CanonicalFixture {

  id: ~ expression "building"

}
```

Transform Canonical Smart Bulb To Fixture Relation To Smart Bulb To Fixture Relation

`TransformCanonicalSmartBulbToFixtureRelationToSmartBulbToFixtureRelation` maps fields in `CanonicalSmartBulbToFixtureRelation` to fields in `SmartBulbToFixtureRelation`.

```
/**
 * This type encapsulates the data flow from the {@link
CanonicalSmartBulbToFixtureRelation} to the {@link
SmartBulbToFixtureRelation} type.
 */
type TransformCanonicalSmartBulbToFixtureRelationToSmartBulbToFixtureRelation
mixes SmartBulbToFixtureRelation transforms
CanonicalSmartBulbToFixtureRelation {

  id:      ~ expression "concat(SN,'_',fixture)"
  to:      ~ expression { id: "fixture" }
  from:    ~ expression { id: "SN" }
  end:     ~ expression "end"
  start:   ~ expression "start"

}
```

Transform Canonical Smart Bulb Event To Smart Bulb Event

`TransformCanonicalSmartBulbEventToSmartBulbEvent` maps fields in `CanonicalSmartBulbEvent` to fields in `SmartBulbEvent`.

```
/**
 * This type encapsulates the data flow from the {@link
CanonicalSmartBulbEvent} to the {@link SmartBulbEvent} type.
 */
type TransformCanonicalSmartBulbEventToSmartBulbEvent mixes SmartBulbEvent
transforms CanonicalSmartBulbEvent {

  id:        ~ expression "id"
  end:       ~ expression "end"
  start:     ~ expression "start"
  eventCode: ~ expression "eventCode"
  eventType: ~ expression "eventType"
  smartBulb: ~ expression { id: "bulb" }

}
```

Transform Canonical Power Grid Status To Building

`TransformCanonicalPowerGridStatusToBuilding` maps fields in `CanonicalPowerGridStatus` to fields in `Building`.

```
/**
 * This type encapsulates the data flow from the {@link
CanonicalPowerGridStatus} to the {@link PowerGridStatusSet} type.
 */
type TransformCanonicalPowerGridStatusToBuilding mixes Building transforms
CanonicalPowerGridStatus {

  id:          ~ expression "Building"
  gridStatus:  ~ expression { timestamp: "TS", value: "Status"}

}
```

Transform Canonical Smart Bulb To Manufacturer

`TransformCanonicalSmartBulbToManufacturer` maps fields in `CanonicalSmartBulb` to fields in `Manufacturer`.

```
/**
 * This type encapsulates the data flow from the {@link CanonicalSmartBulb}
to the {@link Manufacturer} type.
 */
type TransformCanonicalSmartBulbToManufacturer mixes Manufacturer transforms
CanonicalSmartBulb {

  id:          ~ expression "Manufacturer"

}
```

Provision your application now.

We have finished making canonicals and canonical transforms! You are ready to load data onto the Platform and map it to your data model.

4. LOADING DATA

In this section we show you how to use the `cURL` command to import data into the Platform.

Before loading data, however, we need to ensure we have the right configurations. The `JMSDataLoad` queue is an invalidation queue that manages requests for loading data onto the platform. To enable the `JMSDataLoad` queue, there must exist a `TenantConfig` with `id=JMSEnabled` that has its value set to true.

The following command will either create a new `TenantConfig` or update the existing `TenantConfig` with `id=JMSEnabled`. Type this command into your console:

```
TenantConfig.upsert({id: 'JMSEnabled', value: 'true' })
```

A. The cURL Command

Below is a template `cURL` command you can run in your terminal (not the console!) to upload data, where the situation-dependent fields are highlighted for visibility:

```
curl -v -H "Content-Type: FileType" -X PUT --data-binary @PathToFile
DevelopmentURL/import/1/Tenant/Tag/CanonicalName/File -u 'Username' -p
```

Let's give an example. Consider a user John Doe, who finds himself in this situation:

- User John Doe
  - *Username*: `john.doe@c3iot.com`
  - *Tag*: `johndoec3iotcom`

- Uploading data from `SmartBulbMeasurement.csv` to `CanonicalSmartBulbMeasurement`
  - *FileType*: `text/csv`
  - *File*: `SmartBulbMeasurement.csv`
  - *CanonicalName*: `CanonicalSmartBulbMeasurement`
- **Alternative:** If you are using the compressed (gzipped) `SmartBulbMeasurement.csv.gz` file, you should use these properties to upload data from `SmartBulbMeasurement.csv.gz` to `CanonicalSmartBulbMeasurement.` This is only needed for this file.
  - *FileType*: `text/csv`
  - ***Additional Header***: `-H "Content-Encoding: gzip"`
  - *File*: `SmartBulbMeasurement.csv.gz`

  - *CanonicalName*: `CanonicalSmartBulbMeasurement`

- `SmartBulbMeasurement.csv` is stored in the *Downloads* folder on John Doe's computer
  - *PathToFile*: `/Users/johndoe/downloads/SmartBulbMeasurement.csv`

- User is operating in a particular environment, working on the `lightBulbPM` package
  - *Tenant*: `lightBulbPM`
  - *DevelopmentURL*: `https://johndoec3iotcom-lightBulbPM.c3-e.com`

A curl command for a user in this situation would look like this:

```
curl -v -H "Content-Type: text/csv" -X PUT --data-binary
   @Users/johndoe/Downloads/SmartBulbMeasurement.csv
   https://johndoec3iotcom-lightBulbPM.c3-
   e.com/import/1/lightBulbPM/johndoec3iotcom/CanonicalSmartBulbMeasurement/S
   martBulbMeasurement.csv -u 'john.doe@c3iot.com' -p
```

As you can see, *tenant* is the name of the package we are working on – `lightBulbPM` – and *tag* is your username on the C3 IoT Platform without any special characters (example here is `johndoec3iotcom`).

The *path to file* is the path to the file we are uploading from your computer, in this example `SmartBulbMeasurement.csv`. The *development URL* is a URL that indicates the environment we are working in. Your *development URL* may be different. Remember, C3 IoT Training URLs look like this:

`https://<TagName>-<TenantName>.c3-e.com`

Now that you're more familiar with `cURL`, you are tasked with uploading data from `SmartBulbMeasurement.csv` to `CanonicalSmartBulbMeasurement`. Use the given template, filling in the dependent fields with information relevant to you. Good luck!

B. Monitoring Data Load

To monitor the progress of your data load, you can enter either of the following commands into the console:

```
(1) c3Grid(DataLoadProcessLog.fetch())
(2) c3Grid(DataLoadUploadLog.fetch())
```

These commands will show the status of each upload and help you figure out which loads were successful and which weren't.

You are also encouraged to consult the "Console Cheat Sheet" document, which has commands for exploring queues and the progress of data loads.

C. Checking the cURL Upload

Congrats! Make sure that your `SmartBulbMeasurement.csv` data has been successfully loaded to the Platform by running `c3Grid(SmartBulbMeasurement.fetch())` in the console. You should see several `SmartBulbMeasurement` types appear, matching the data provided in `SmartBulbMeasurement.csv`.