Data Model and C3 Types

CAPSTONE PROJECT | ASSIGNMENT 1

This series of assignments is centered on creating a light bulb predictive maintenance application. The goal of this application is to assist building managers as they manage a network of light bulbs.

To this end, the application is designed to enable two key use cases:

1. Tracking light bulbs distributed throughout multiple buildings.

2. Predicting the likelihood that each light bulb will fail in a given timeframe.

You will be developing the building blocks of this application through the companion exercises included in each content module.

This exercise focuses on the first step: defining a data model using C3 Types.

When designing a data model, it is often helpful to begin by evaluating the available data, then considering how that data logically relates. For this project, we have the following data:

- Static data about each light bulb
  - Serial number, bulb type, and manufacturer, among other fields
- Static data about each building
- Static data about apartments within those buildings
- Static data about which fixtures are in each apartment
- Time series data on power grid by building
- Time series data on which light bulbs are in which fixtures over time
- Time series data on sensor measurements for each smart bulb
  - 15-minute data on features like voltage, wattage, on/off status etc.
  - Event data on smart bulbs (i.e. connectivity issues, overheating)

These data sources combine to form a fairly comprehensive picture of our network.

How are these data sources logically related? At the center of our data model is the light bulb. We then extend the light bulb to incorporate sensor and event data, creating the smart bulb. Smart bulbs can be placed in fixtures. These fixtures are located in apartments which are located in buildings.

0. BEGINNING NOTES

Please download and use the Google Chrome browser for all provisioning tasks.
For code writing, please use an IDE like Eclipse, VS Code, or Sublime.

As you build applications you will inevitably encounter errors and make mistakes. What you learn from this course is not only the fundamentals of the C3 IoT Platform, but how to troubleshoot problems and resolve issues. This skill is incredibly valuable.

We've made effort to have clear error messages and reporting of bugs so that you can understand how to progress when you hit an obstacle. There are many places you may be alerted of errors such as in the console, in the provisioner, and during data loading.

Whenever you encounter an error, follow these steps:

1. Read the error message to locate the source of the problem. Oftentimes the message will tell you exactly what the problem is, and it will be an easy fix.

2. Using the C3 documentation, console, Capstone Project instruction sheets, course lectures, and "Console Cheat Sheet" try to figure out what is wrong with whatever the error message pointed you towards.

3. If you are still unable to figure out what went wrong, post a question on https://community.c3iot.ai/. Be sure to include a detailed description of what you were doing as well as the error message text and any other relevant information.

With diligence, you will finish this project with a solid understanding of how to build an application on the C3 IoT Platform and will be armed with the confidence to fix errors as they arise.

1. GETTING STARTED

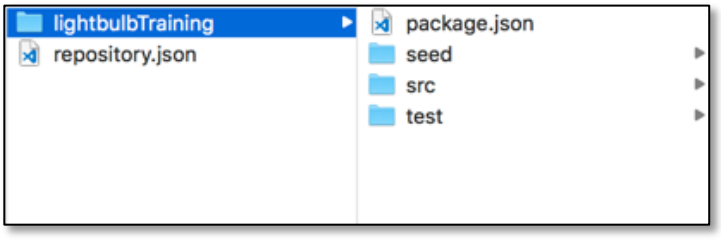
A. Creating the Application

First, download and unzip "IDE Starter Kit.zip."

This folder contains the basic files as well as the folder structure you'll need to get started.

Your root folder, named `/training` in the "IDE Starter Kit," includes a `package.json` file that specifies the packages your package depends upon. Simply put, these packages are imported into your package and can be overwritten and extended as you choose.

Below is an image illustrating what the `/training` folder contains:



We save all C3 Types (`.c3typ` and implementation files) in the `/src` folder. Inside that folder, you can use any structure you like.

Some folders have exact naming and nesting requirements that must be met to avoid provisioning errors, but we will discuss these later.

B. Provisioning the Application

Provisioning is deploying code to a target environment. In this process, several checks are executed to ensure correctness. The process will alert you if errors are detected and abort the deployment.

To see your application working in your target environment, you will need to provision your tenant – which in this context refers to your code package – to a tag on the environment. If the tag you have specified does not exist, it can be created during provisioning.

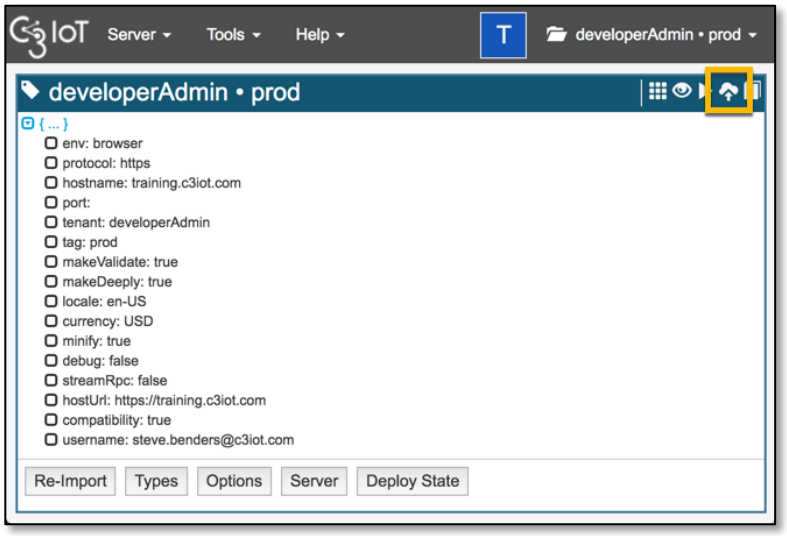
If you have installed `c3-prov`, you can provision code from your IDE or command line.

You can also provision using your environment's console by visiting the appropriate URL. This URL varies by environment and deployment. C3 IoT training URLs are often formed in the following way:

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

The `/static/console` at the end of the URL indicates that we are navigating to the provided interface to interact with our environment. We call this the "console". We will explore the console more in a later section.

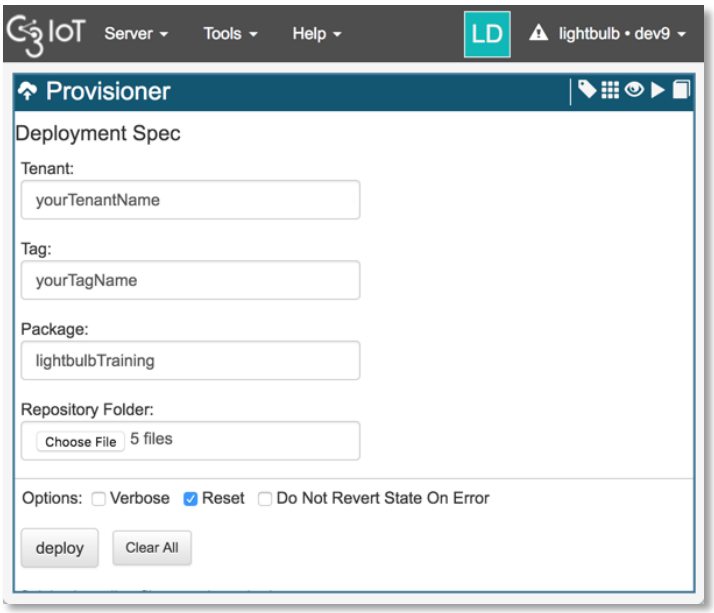
Once the page has loaded, navigate to the Provisioner tool by clicking the cloud-and-arrow icon in the top right:



In the Provisioner, specify the following fields:

- `Tenant: yourTenantName`
- `Tag: yourTagName`
- `Package: lightbulbTraining`
- `Repository folder: training/`

Tenant and tag name vary. With the "IDE Starter Kit" folder, the package name is `lightbulbTraining` and the repository folder (the folder that contains `/src` and the `repository.json` file) is `/training`.



Checking the "Reset" box means that the deployment will be fully regenerated using your code. Leaving it unchecked only deploys updates to the existing deployment.

Deployment may take up to a few minutes. If Chrome prompts you to kill the page, select "Wait" to allow the deployment to continue.

If errors are detected in the code, they are displayed below the buttons and the deployment is aborted.

To interact with your new deployment via the console, either refresh the browser or execute the `c3ImportAll()` command.

2. CREATING C3 TYPES

Based upon the data model we described earlier, we will create six types:

1. `LightBulb`
2. `SmartBulb`
3. `Building`
4. `Fixture`
5. `Apartment`
6. `SmartBulbEvent`

All of these types must be `persistable` and require schema names. Schema names are uppercase, less than eighteen characters, and use alphabetical characters and underscores.

A type declaration, including a schema name, is written as follows:

```
// Entity type makes the type persistable.
entity type LightBulb schema name "LGHT_BLB"
```

Type fields can be primitives or references (i.e. `string`, `SmartBulb`, or `[Fixture]`).

Your task is to create the six types enumerated above, including the necessary reference and collection fields. Remember to save these `.c3typ` files to the `/src` folder.

Light Bulb

We will begin by creating the `LightBulb` type, since it is our data model's central type.

On this type, we define the technology this light bulb uses, its manufacturer, its wattage, and its start date as follows:

```
/**
 * LightBulb.c3typ
 * A single light bulb.
 */
extendable entity type LightBulb schema name "LGHT_BLB" {

  // The type of this light bulb.
  bulbType: string enum('LED', 'INCAN', 'CFL')

  // The light bulb's wattage
  wattage: decimal

  // The time at which this light bulb was manufactured.
  startDate: datetime

}
```

Smart Bulb

We will now extend the `LightBulb` type to include measurement and geographic data.

```
/**
 * SmartBulb.c3typ
 * A bulb capable of storing power consumption, light output, location, and
more.
 */
entity type SmartBulb extends LightBulb mixes MetricEvaluatable type key
"SMRT_BLB" {

  // The latitude of this bulb.
  latitude: double


  // The longitude of this bulb.
  longitude: double

  // The unit of measurement used for this bulb's light output measurements.
  lumensUOM: Unit

  // The unit of measurement used for this bulb's power consumption
measurements.
  powerUOM: Unit

  // The unit of measurement used for this bulb's temperature measurements.
  temperatureUOM: Unit

  // The unit of measurement used for this bulb's voltage measurements.
  voltageUOM: Unit

}
```

Mixing `MetricEvaluatable` into the above type does exactly what the name suggests: it enables us to evaluate metrics written on the `SmartBulb` type.

Building

The `Building` type contains apartments, which contain fixtures and smart bulbs.

```
/**
 * Building.c3typ
 * A single building containing many {@link Apartment}s.
 */
entity type Building mixes MetricEvaluatable schema name "BLDNG"
```

Fixture

The `Fixture` type, of course, contains smart bulbs.

```
/**
 * Fixture.c3typ
 * A single fixture in which a single {@link SmartBulb} may be connected.
 */
entity type Fixture mixes MetricEvaluatable schema name "FXTR" {

  // The {@link Apartment} in which this fixture is located.
  apartment: Apartment

}
```

Smart Bulb Event

The `SmartBulbEvent` type contains information on events that happen to smart bulbs.

```
/**
 * SmartBulbEvent.c3typ
 * An event that happens to a single {@link SmartBulb}.
 */
entity type SmartBulbEvent schema name "SMRT_BLB_EVNT" {

  // The time at which this event ended.
  end: datetime

  // The time at which this event began.
  start: datetime

  // The event code used to distinguish events.
  eventCode: string

  // The type of this event.
  eventType: string

  // The {@link SmartBulb} connected to this event.
  smartBulb: SmartBulb

}
```

Manufacturer

The `Manufacturer` type stores data on bulb manufacturers. We'll use it in a future assignment.

```
/**
 * Manufacturer.c3typ
 * A company that manufactures lightbulbs.
 */
entity type Manufacturer mixes MetricEvaluatable schema name "MNFCT"
```

3. IMPROVING THE DATA MODEL

We added several simple reference fields in the previous section, but the data model is not yet fully connected. We must now add fields to several of our types.

Light Bulb

Now that we have a `Manufacturer` type, we can reference it from `LightBulb`.

```
/**
 * LightBulb.c3typ
 * A single light bulb.
 */
extendable entity type LightBulb schema name "LGHT_BLB" {

  // The type of this light bulb.
  bulbType: string enum('LED', 'INCAN', 'CFL')

  // The light bulb's wattage
  wattage: decimal

  // NEW FIELD: The name of this light bulb's manufacturer.
  manufacturer: Manufacturer

  // The time at which this light bulb was manufactured.
  startDate: datetime

}
```

Smart Bulb

Since we also now have `SmartBulbEvents`, we'll want to correlate events with smart bulbs. Because multiple events can correlate with a single smart bulb, we'll use an array reference.

```
/**
 * SmartBulb.c3typ
 * A bulb capable of storing power consumption, light output, location, and
more.
 */
entity type SmartBulb extends LightBulb mixes MetricEvaluatable type key
"SMRT_BLB" {

  // The latitude of this bulb.
  latitude: double

  // The longitude of this bulb.
  longitude: double

  // The unit of measurement used for this bulb's light output measurements.
  lumensUOM: Unit

  // The unit of measurement used for this bulb's power consumption
measurements.
  powerUOM: Unit

  // The unit of measurement used for this bulb's temperature measurements.
  temperatureUOM: Unit

  // The unit of measurement used for this bulb's voltage measurements.
  voltageUOM: Unit

  // NEW FIELD: This bulb's historical events.
  bulbEvents: [SmartBulbEvent](smartBulb)

}
```

Building

Each building can house multiple apartments, so we'll use an array reference field again here.

```
/**
 * Apartment.c3typ
 * A single building containing many {@link Apartment}s.
 */
entity type Building mixes MetricEvaluatable schema name "BLDNG"{

  // NEW FIELD: The apartments that are inside this building.
  apartments: [Apartment](building)

}
```

Apartment

Each apartment can contain multiple fixtures.

```
/**
 * Apartment.c3typ
 * A single apartment unit in a {@link Building}.
 */
entity type Apartment mixes MetricEvaluatable schema name "APRTMNT" {

  // The {@link Building} in which this apartment is located.
  building: Building

  // NEW FIELD: An array of fixtures that exist in this apartment.
  fixtures: [Fixture](apartment)

}
```

At this point, you have created the basic data model for the entire capstone project.

Before continuing, provision your package to an environment to confirm there are no errors.

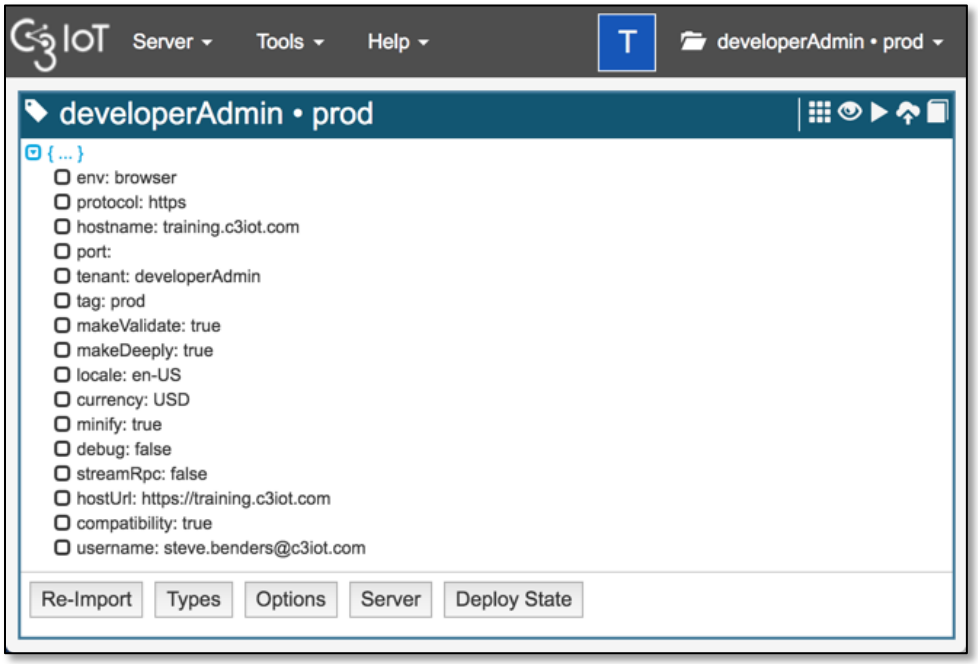
4. EXPLORING TYPES IN CONSOLE

After provisioning your package, visit the console to examine your data model.

You may remember from earlier that you can visit the console at a URL formatted as follows:

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

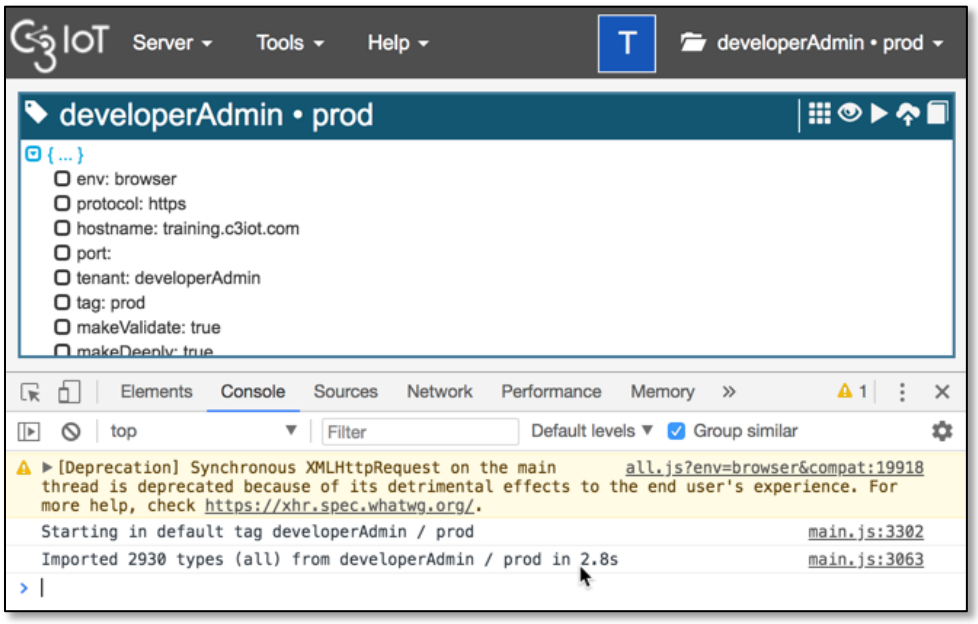
At this URL, you should see something like the following:



In Chrome, you can bring up the JavaScript console using a shortcut as follows:

- Mac: “Command + Option + J”
- Windows: “Control + Shift + J”

You can also bring up the JavaScript console by clicking “View > Developer > JavaScript Console.” Regardless of which approach you choose, you should now see the following:



From the JavaScript console, you can execute the following commands:

```
// Refresh tags and types.
c3ImportAll()

// Explore type definitions.
c3ShowType(SmartBulb)

// Define variables.
SmartBulb.make(...) // Temporary only – not persisted when the browser is
closed.
SmartBulb.create(...) // Persisted to the database.
var bulb1 = SmartBulb.create({id: "bulb-1", bulbType: "LED", startDate:
"2017-01-01T08:00:00.000Z"})

// Use the merge command to modify select fields while preserving other
fields.
bulb1.manufacturer = "Philips"
SmartBulb.merge(bulb1)

// Fetch and get – the main ways to retrieve persistable objects from the
database.
var anotherBulb1 = SmartBulb.get("bulb-1") // Retrieves a single object by
id.
c3Grid(anotherBulb1) // Visually displays information about the object.
c3Grid(SmartBulb.fetch()) // Fetch ≥ 0 instances of one type in the objs
field.
```

The `fetch()` call is incredibly common and useful. It takes a `FetchSpec`, which allows you to filter on a number of fields:

- `include`: Defines which fields we want returned
  - Specified using a comma-separated string
  - This field can include nested reference objects
- `filter`: Determines which objects should be returned
- `order`: Determines the sorted order in which objects should be returned
- `limit`: Limits the maximum number of objects returned (`-1` means unlimited)

Try out the following command to retrieve and display `SmartBulb` bulb type and manufacturer data for bulbs that are made by Philips, ordered by `id`:

```
c3Grid(SmartBulb.fetch({include: "bulbType, manufacturer", filter:
"manufacturer == 'Philips'", order: "descending(id)", limit: -1}))
```

Since we'll be uploading real data in the next sections, be sure to remove all `SmartBulb` data you may have generated using one or more of the following commands:

```
// The .remove() call can accept the id as a string or an object with an id
field.
SmartBulb.remove("bulb-id")

// Remove all takes no arguments and deletes all data for the specified type.
SmartBulb.removeAll()
```