



Mobile Device Management System for a Telecom Provider

A Use Case For



express serverless
platform



Background

A multinational conglomerate owns a telecommunication provider and manufacturer subsidiary that operates in all markets throughout the world. The consumer market for mobile devices is crowded and the need to meet customer expectations for features and functions is every growing and accelerating. The need to innovate hardware also requires the ability to deliver these new features and functions over the air (OTA) to keep up with security patches, new features and marketing campaigns and promotions.

Telecommunications industry impact

Telecommunication providers have the need to provision digital goods and services through an increasing number of devices. Devices are no longer necessarily centrally owned and distributed by the telecommunication provider. The consumerization of telecommunications in general has lead to interoperability requirements based on industry protocols and standards that require more sophisticated device management capabilities.

Core imperative

To be able to get a customer registered within a network and to maintain a consistent high quality level of service with updates, the telecommunication provider needed to create a new generation of mobile device management system (MDM). The MDM system would be built on top of a set of microservices so that it could be continuously evolved and also be leveraged for both internally among the telecommunication provider's business units as well as externally with the telecommunication provider's supply chain providers and their go to market retail partners.

Customer's impact

The telecommunication provider's ability to provision and manage mobile devices numbering in the billions for both current and previous generation models is constrained by not only the company's resources but also by the fact that the ongoing service aspect of such devices is not their core business focus. By partnering and empowering their partners with a MDM system that gives them centralized control but distributed operation, the provider can ensure a high level of service and scale with their end user's demand.

Legacy challenge

The existing MDM capabilities at the telecommunication provider still services the previous generation devices that are still in use by its earliest customers. The existing system has a rich set of data including: user profile, device profile and network settings that are customized for each class of user and each class of device. This data and operation would still need to be utilized in the new MDM capabilities and be made available to the provider's partners who would take over the provisioning and service maintenance aspects of the business that is growing rapidly.

Domain Driven Development

LunchBadger and the telecommunication provider utilized domain driven development to simply and accelerate the development of the MDM system. The MDM system consisted of a new set of microservices that utilized legacy data and entities within the application domain. Each entity was developed as simple model based functions. The functions would have methods to emulate the different actions and behavior of each entity as "models". The application domain for the MDM system is described in more detail below:

Mobile devices can be reconfigured over-the-air after it has been shipped out of a factory. The system may use a cloud-based server to store configuration objects (binaries) periodically published by device vendors. In this domain the entities of interest could be device (characteristics), vendor profiles, configuration objects, etc. (We will get into the details of such a system in following sections.)

It may be worth recalling that all the microservices in an application need not use the same data store or persistence mechanism. So also, the entities in our domain may be mapped to [different databases](#), depending on the data structures and data access requirements involved. Domain driven development does not imply a monolithic data layer.

Device Provisioning System

The following is a walkthrough of the provider's MDM system built using LunchBadger. Actual references including entities and property names have been changed for confidentiality purposes.

The MDM system will be referred to as the "Device Provisioning System". It will enable device vendors to apply configuration settings to mobile devices, months after it has been shipped out of the factory. For example a vendor could apply over-the-air configuration settings to turn a mobile device into a point-of-sale terminal, or a survey response collector or a digital signage.

The central component of this system will be a **Device Manager** that exposes APIs for:

- Vendors to upload configuration settings (probably self-installable binaries) for one or more models of mobile devices. Actually apart from the device vendors, there may be third party ecosystem players, called Partners, who offer such device reconfiguration capabilities.
- Device users to register / enroll with the device manager so that available configuration settings can be retrieved and installed on demand

Considering the problem domain we can identify at least three entities:

- **PartnerDevice**: Actually all it means is a way to associate a configuration setting with the provider of the configuration capability, or a Partner. This entity should include a model name, that is a device model to which the configuration setting can be applied.
- **UserAgent**: This will represent the mobile device itself, which can be configured over the air. Therefore, information like make, model, manufacturer, IMEI number, and probably the current location of the device (latitude and longitude) may be captured in this entity.
- **ConfigObject**: This is the actual binary object that can be downloaded and probably auto-installed on a mobile device, based on user preferences.

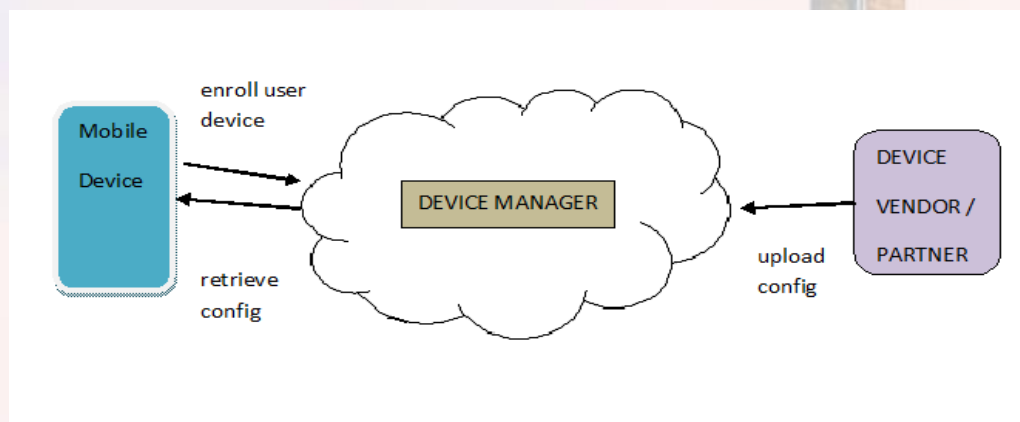


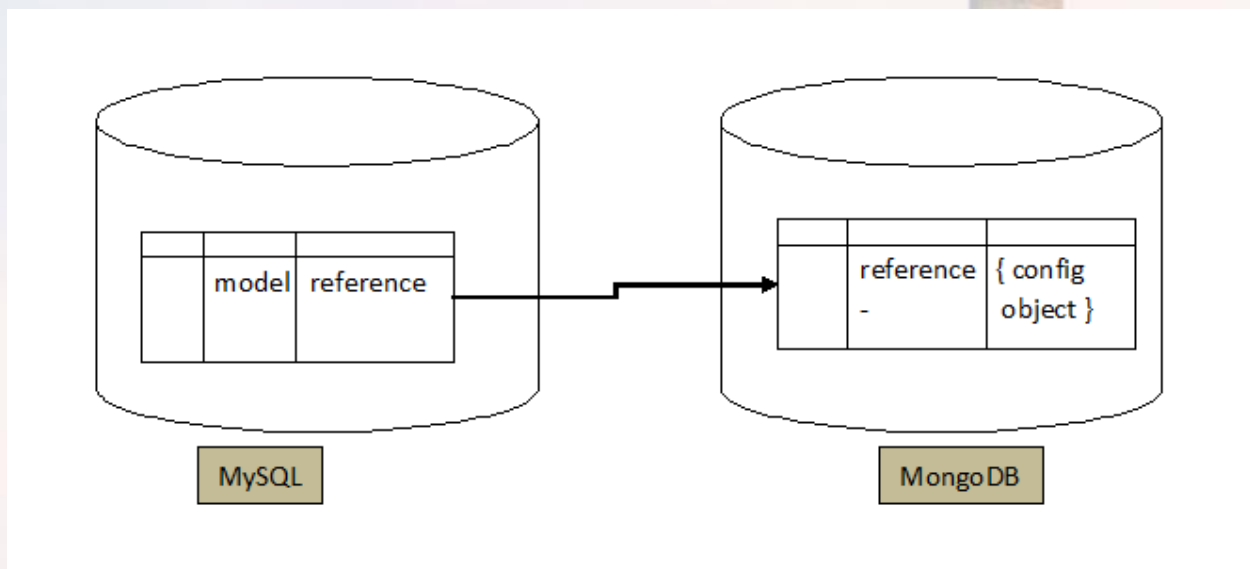
Figure: DEVICE PROVISIONING SYSTEM - We will build this application on top of the Express Serverless Platform.

Step-by-step Implementation

Let's first set up the connector for storing some of the data in our problem domain.

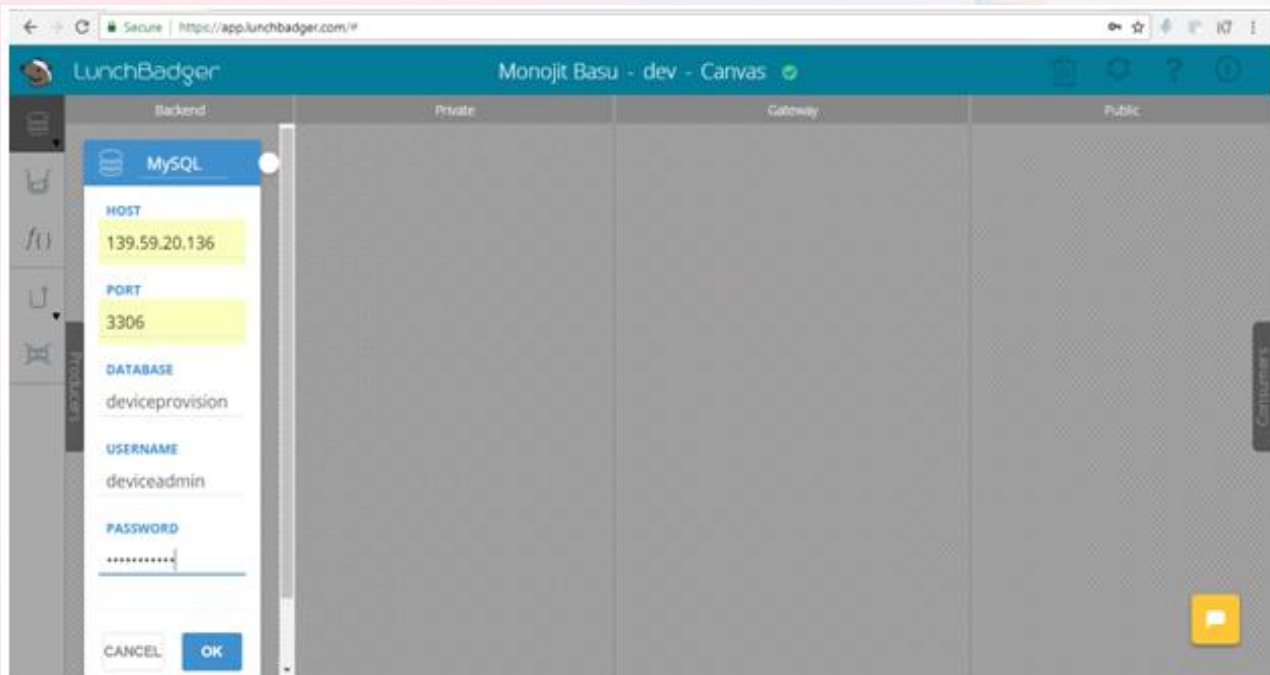
Quite often the individual microservices that constitute an application use different data persistence mechanisms or data stores. To drive home this capability, we will store our data as follows:

The PartnerDevice entity primarily maps a device model to a configuration object. But wait. Isn't the configuration object more like semi-structured or unstructured data / a binary string? Well it can be stored in an RDBMS, but to make it more interesting, we will store the configuration object in MongoDB. So, in a full-fledged application, the configuration objects could be JSONs or BSONs (in MongoDB). So, the PartnerDevice entity will consist of a device model mapped to a 'reference' string. There will be a corresponding configuration object mapped to this 'reference' string in MongoDB. The 'reference' string will be like a foreign key, although across an RDBMS and a NoSQL, since we have conveniently decided to store the actual configuration objects in a NoSQL.

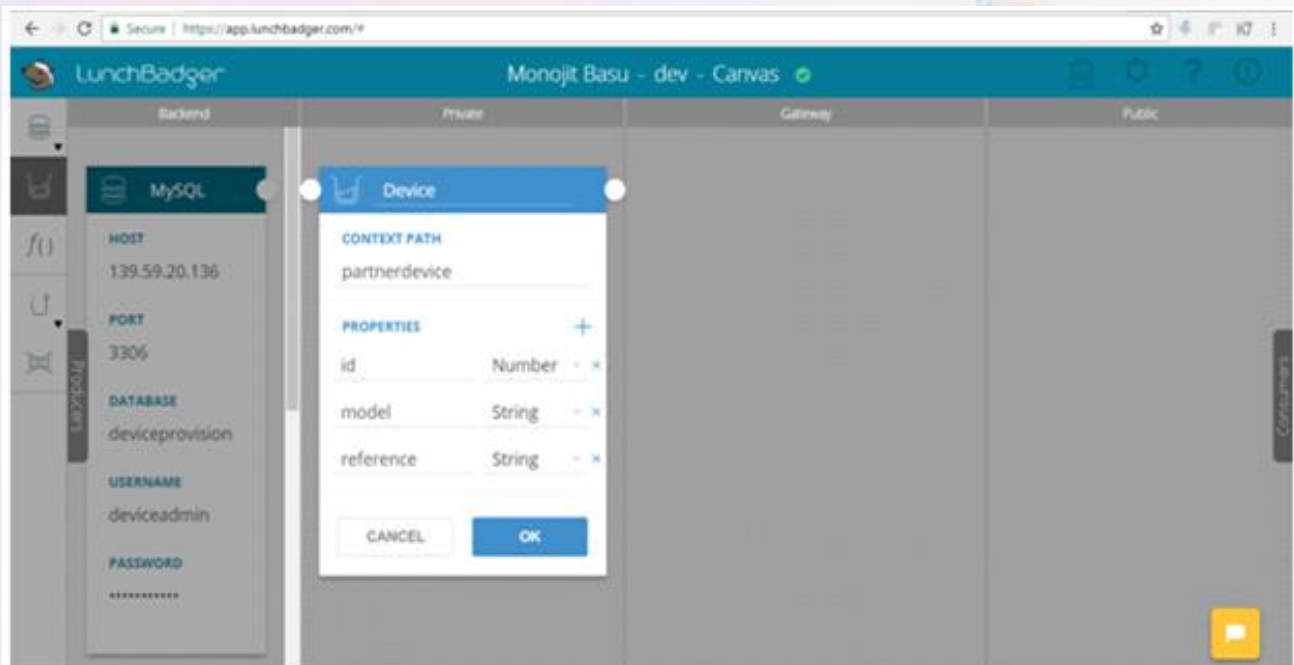


Device Model

Starting with the PartnerDevice entity first, let us set up a MySQL connector. Once we have launched the Canvas, we can drop in a MySQL connector on the 'Backend' quadrant. We will need to supply the connection parameters (in our case, the database password happens to be 'deviceadmin', and the other parameters are visible in the screenshot below).

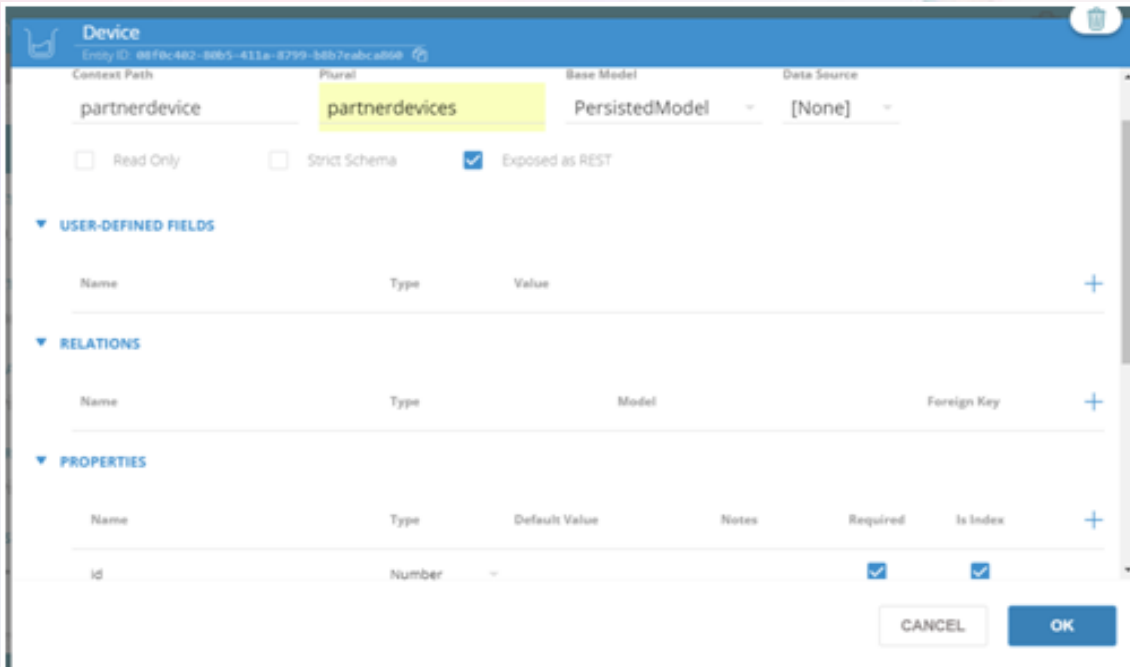


Now we can drop in a Model into the 'Private' quadrant. We want to represent a PartnerDevice entity, and we have already created a table simply called 'Device' in MySQL. We need to map the fields of this table to fields in the Model in our Canvas, and assign correct data types. The data related to this Model will have to be exposed at a certain context path. By default the context name will be the same as the Model name, but we are free to choose a more user-friendly context name like 'partnerdevice'.



Tip: It is a good practice to expand the Model Details and fill in a few more specifics like the plural of the entity we are talking about, whether a data attribute is an index and is required, etc.

For more details feel free to browse through the [platform documentation](#).

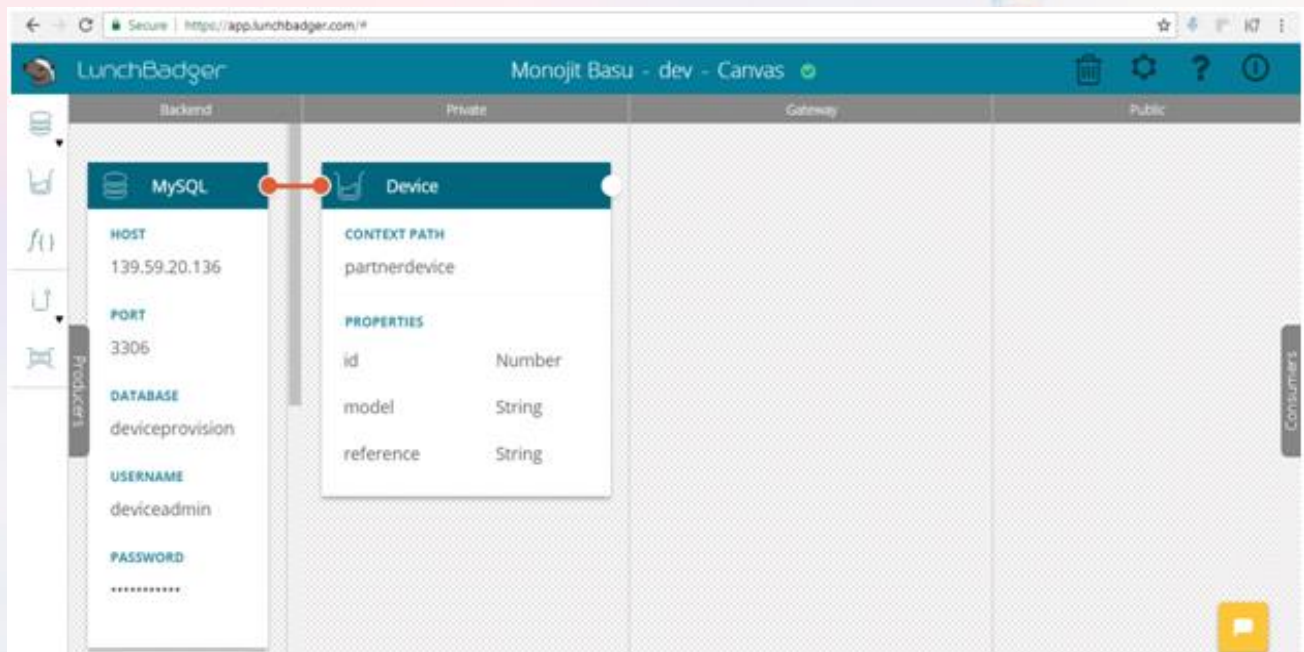


The screenshot shows the configuration interface for a model named "Device". The interface includes the following sections:

- Context Path:** partnerdevice
- Plural:** partnerdevices (highlighted in yellow)
- Base Model:** PersistedModel
- Data Source:** [None]
- Options:** Read Only, Strict Schema, Exposed as REST
- USER-DEFINED FIELDS:** A table with columns: Name, Type, Value, and a plus sign (+).
- RELATIONS:** A table with columns: Name, Type, Model, Foreign Key, and a plus sign (+).
- PROPERTIES:** A table with columns: Name, Type, Default Value, Notes, Required, Is Index, and a plus sign (+). The "Id" property is listed with Type "Number", Required checked, and Is Index checked.

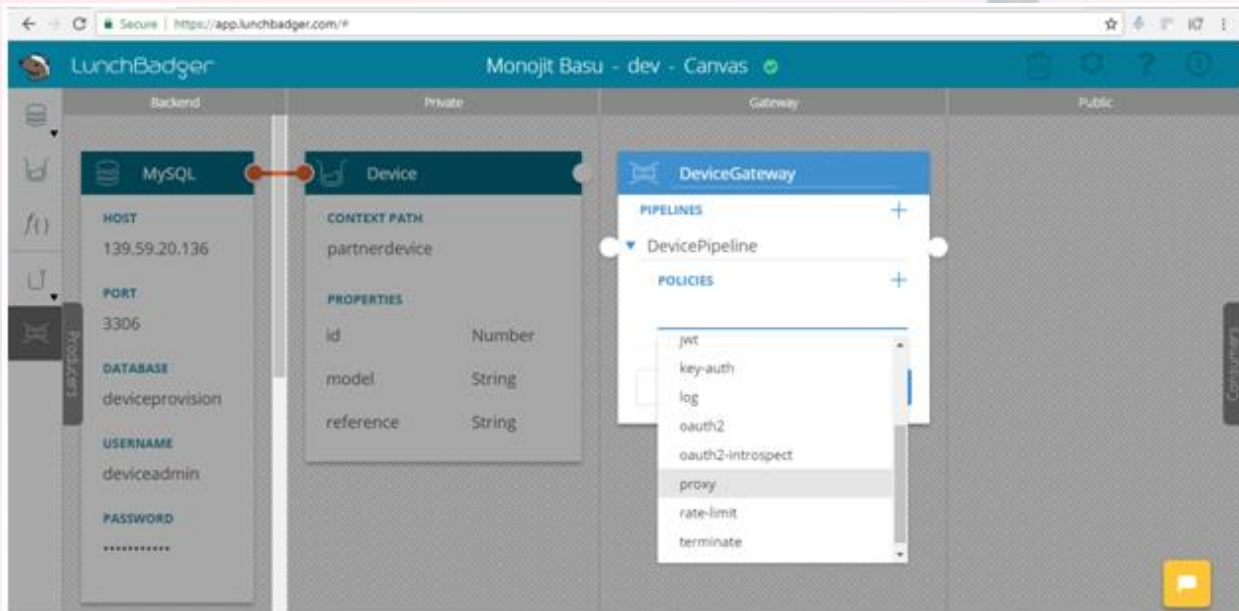
At the bottom right, there are "CANCEL" and "OK" buttons.

Now comes the crucial step: we need to visually connect the Model to the Back-end.



The above steps creates a logical data Model and a data store for it, but we can't yet access the data using RESTful APIs. To do so, we first need to create a Gateway.

Let's just name it 'DeviceGateway'. For our 'Device' model, we need to create a pipeline named 'DevicePipeline' with request processing elements. For now, let's just add a 'proxy'.



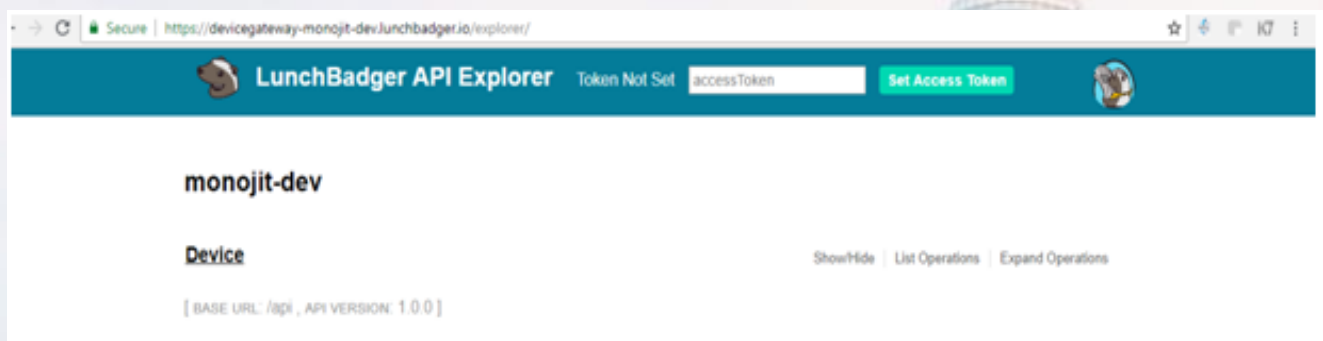
We should see a status message like 'Device Gateway successfully deployed'.



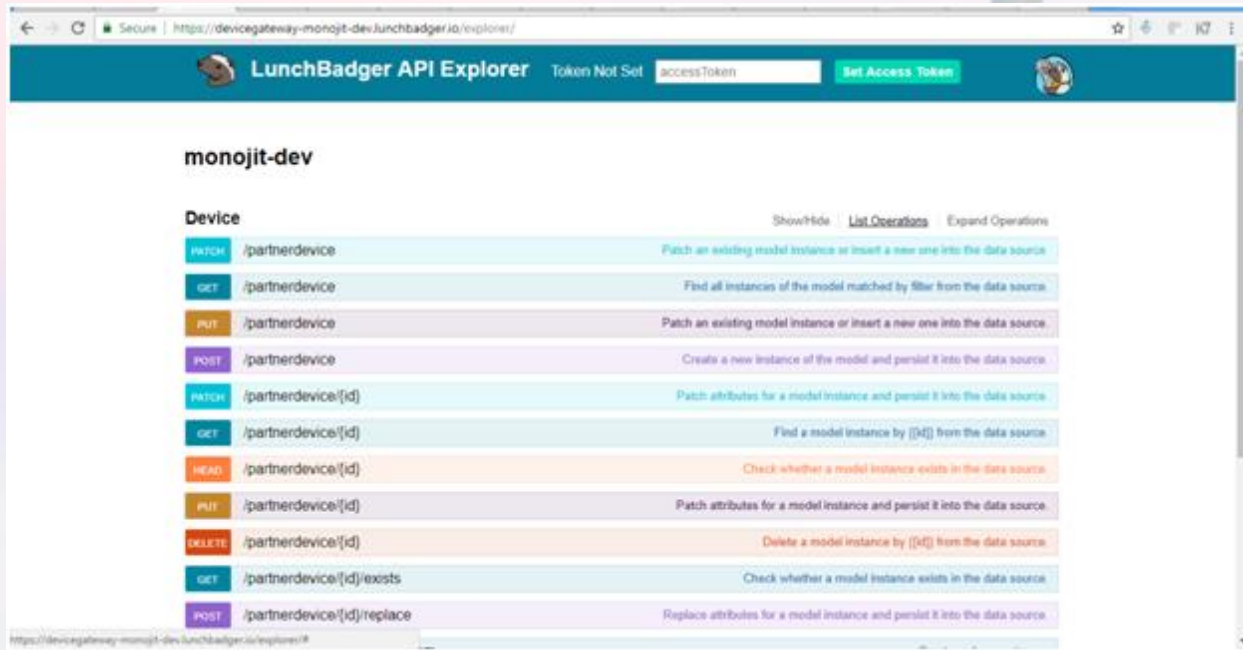
Now one more crucial connecting step: let's connect the 'Device' Model to the 'DevicePipeline'.

Since this is the first time we are connecting the 'Device' Model to a Pipeline in our API Gateway, we will be prompted to create a publicly accessible Endpoint tied to this pipeline. Let's name this as 'DeviceApiEndpoint'. Therein, we can define the URL patterns that we want to be accessed through this endpoint. In this case we want to expose the URLs like '/partnerdevice/*'

Once we complete the above steps, a large set of REST URLs are automatically exposed, based on the underlying data model. To inspect these URLs, click on the 'Settings' button on the top right of the Canvas, and click on the 'explorer' URL.



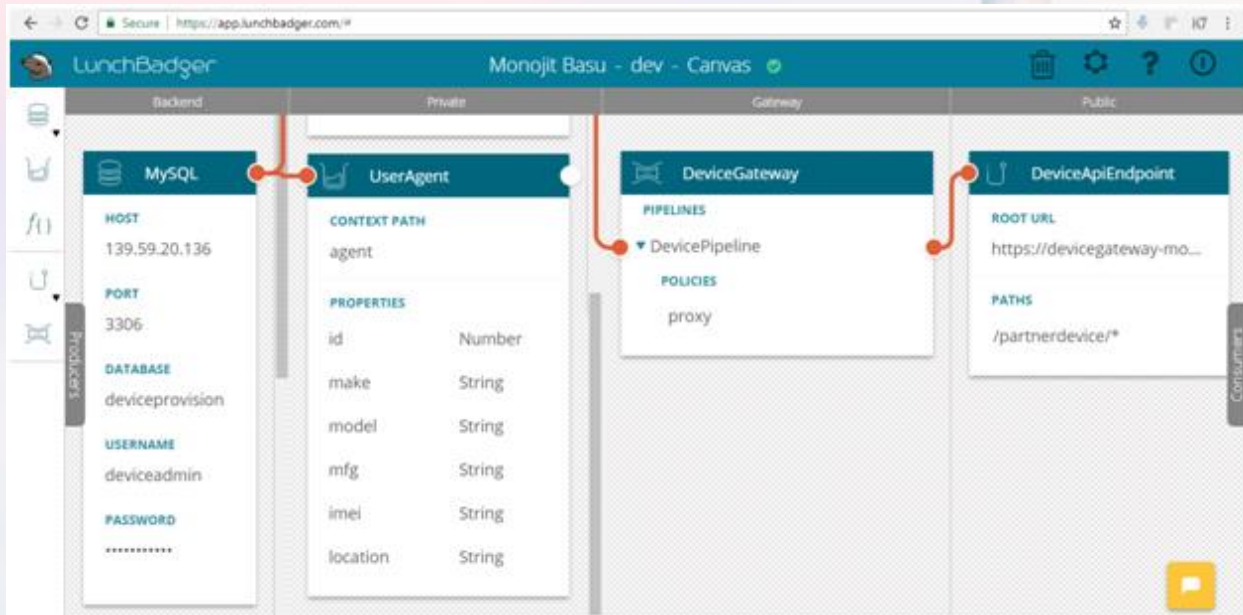
You can expand the list to check out the REST URLs corresponding to our 'Device' Model. Note that the URLs all refer to our chosen context path: '/partnerdevice'.



The Device Model only creates 'references' to configuration objects, but we'll create the configuration objects themselves in MongoDB in a bit.

UserAgent Model

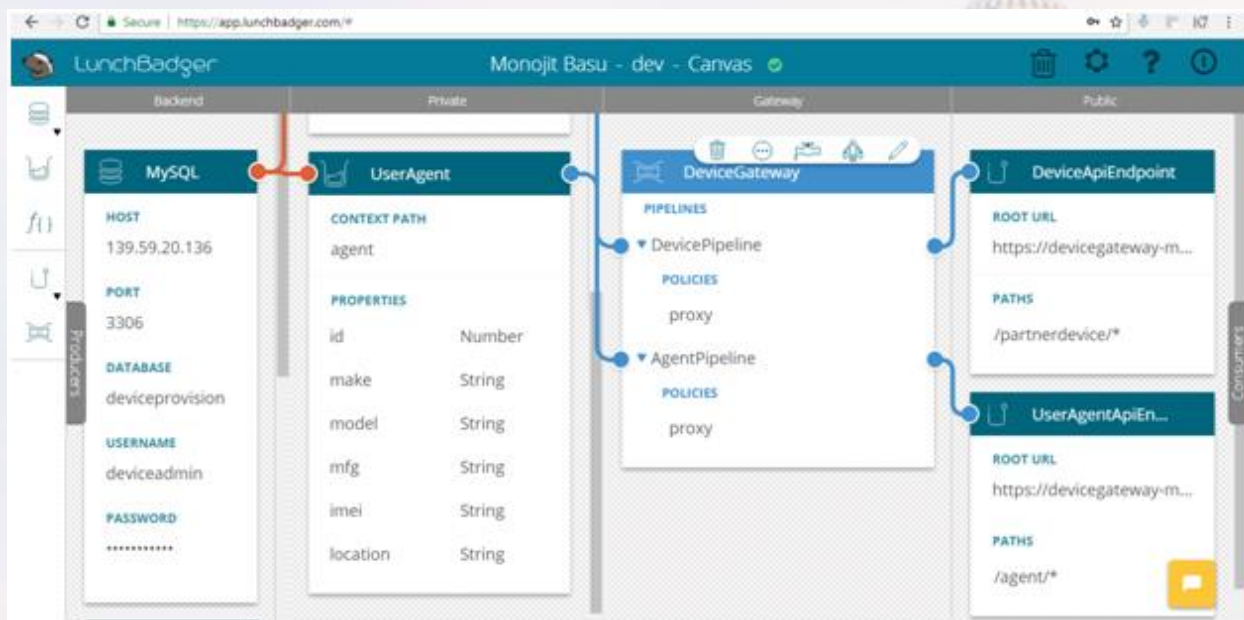
Before that, let's define the 'UserAgent' Model representing mobile devices belonging to users. We'll use the same MySQL back-end. For simplicity, location will consist of latitude and longitude concatenated into a single String. Let the context path for this Model be 'agent'.



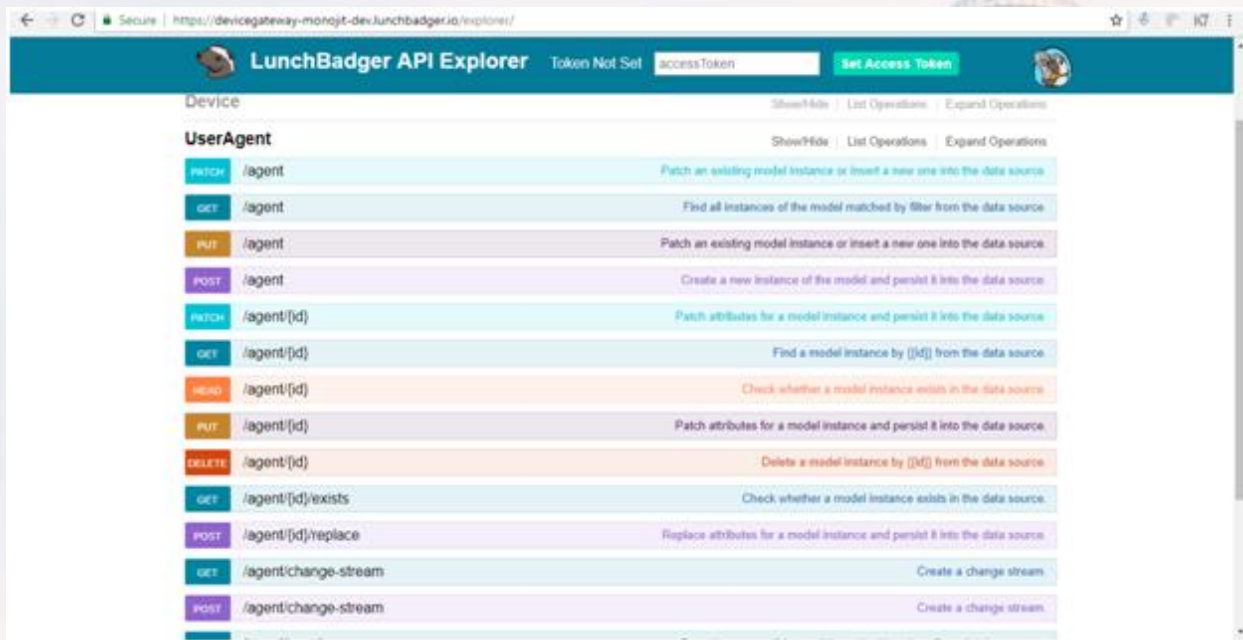
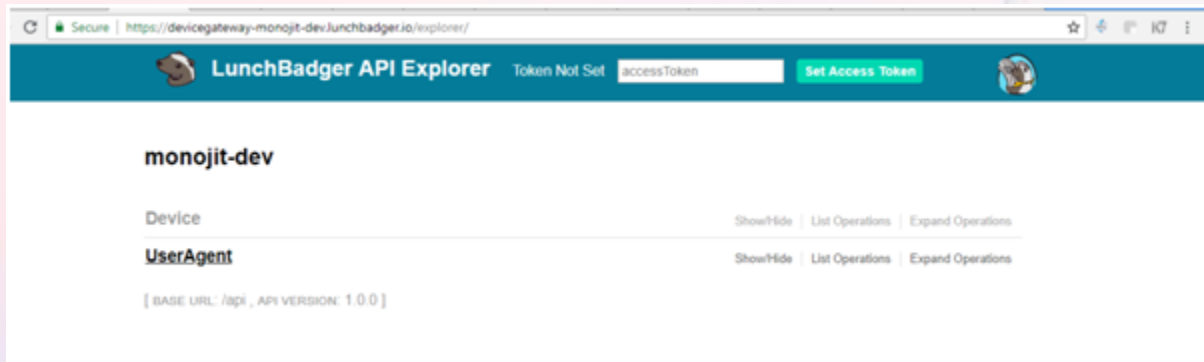
The screenshot shows the LunchBadger management console interface. The top navigation bar includes the LunchBadger logo, the user name 'Monojit Basu', the environment 'dev', and the application 'Carvas'. The main content area is divided into four sections: Backend, Private, Gateway, and Public. The 'Backend' section shows the MySQL database configuration with fields for HOST (139.59.20.136), PORT (3306), DATABASE (deviceprovision), USERNAME (deviceadmin), and PASSWORD (masked). The 'Private' section shows the 'UserAgent' model configuration, including the CONTEXT PATH (agent) and a list of PROPERTIES: id (Number), make (String), model (String), mfg (String), imei (String), and location (String). The 'Gateway' section shows the 'DeviceGateway' configuration, including PIPELINES (DevicePipeline) and POLICIES (proxy). The 'Public' section shows the 'DeviceApiEndpoint' configuration, including the ROOT URL (https://devicegateway-mo...) and PATHS (/partnerdevice/*). Red lines connect the MySQL database to the UserAgent model, and the DeviceGateway to the DeviceApiEndpoint.

Correspondingly, we will create a new pipeline called 'AgentPipeline' exclusively for the UserAgent Model, with just a proxy element within. This will result in one more Endpoint which we will name 'UserAgentEndpoint'. It will expose URLs like '/useragent/*'.

In this example, we will create a separate pipeline in our API Gateway for each Model-based microservice. That way each microservice can be independently configured with a set of pipeline elements (for authentication, authorization, etc) that are most appropriate. This is often the case in many applications, but let us be aware that it is possible to map all our Models to the same Pipeline, if they can all share the same pipeline configuration.



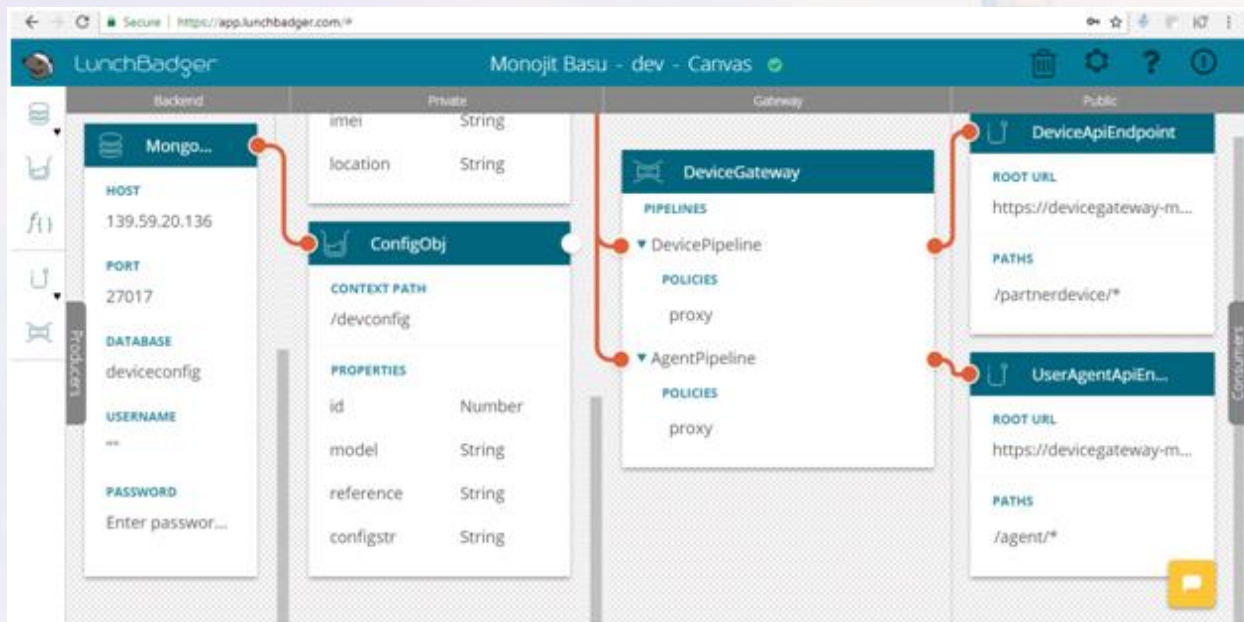
The auto-generated REST URLs can be inspected as follows:



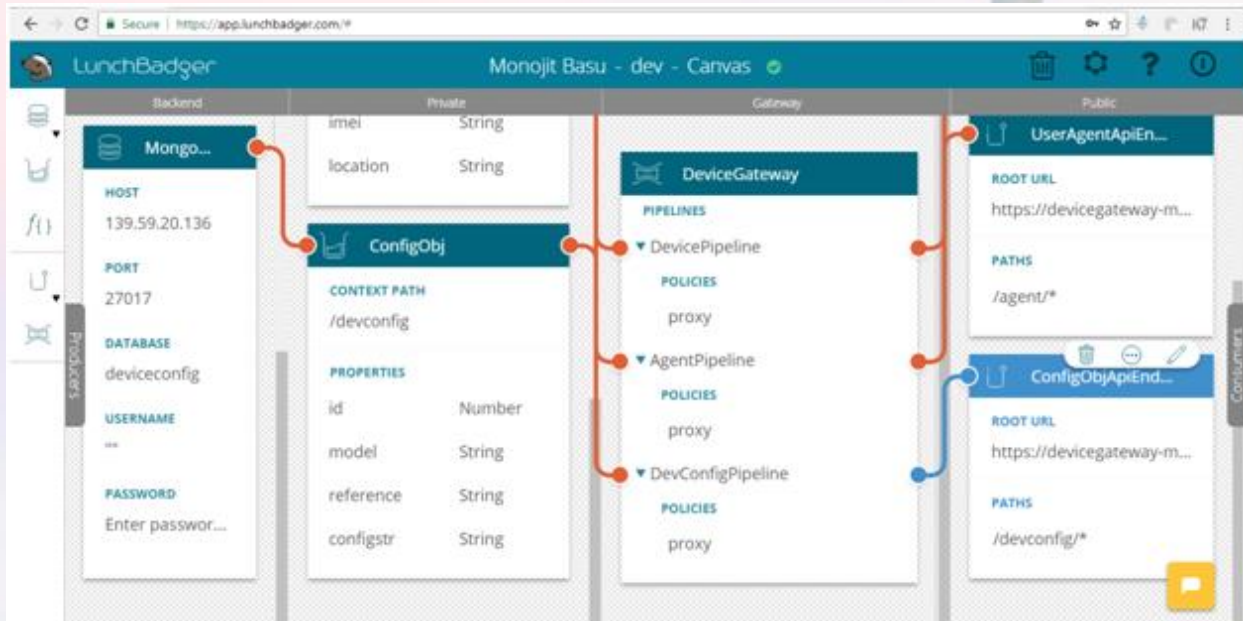
ConfigObj Model

Finally we want to create a Model for ConfigurationObject entities, and store them in MongoDB. Our platform supports a MongoDB connector as well - off the shelf. We'll drop it into the 'Backend' quadrant, and then configure the database connection. Please note that in a production deployment, access to MongoDB should be protected by a username and password.

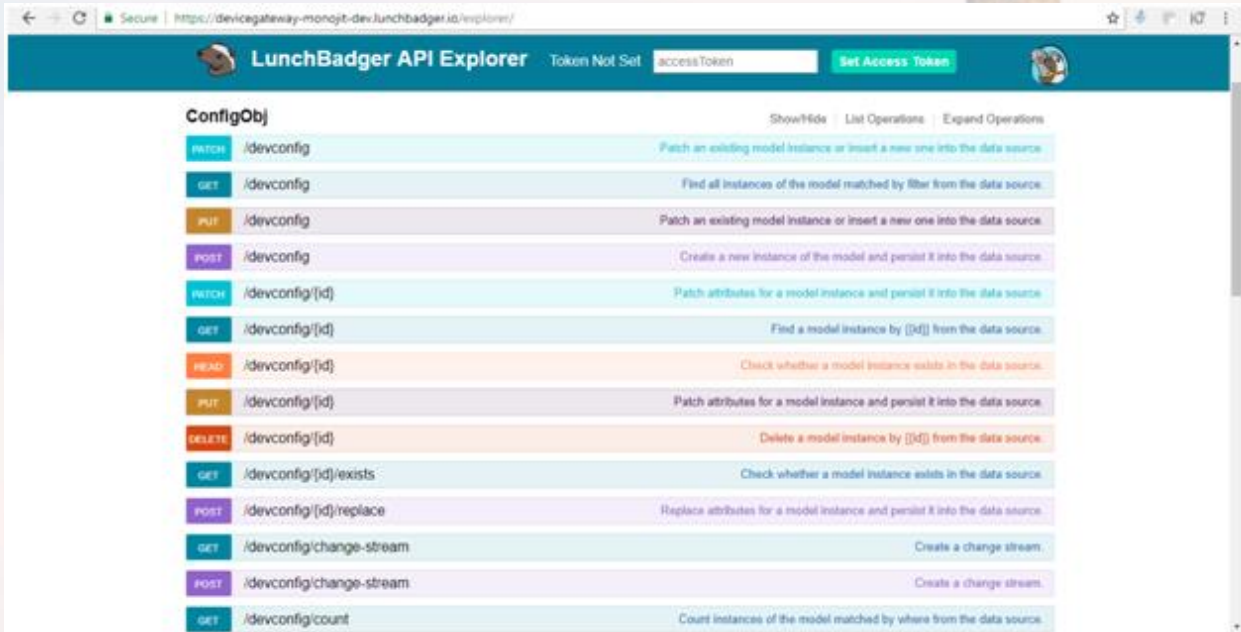
The fields in our 'ConfigObj' Model include a 'model', that is a device model, and a configuration string (a binary string, although in practice it will be more complicated, so it makes sense to store in a NoSQL).



Next we create the 'DevConfigPipeline' within our Gateway and the endpoint named 'ConfigObjApiEndpoint' exposing URLs like '/devconfig/*'.



There will be a bunch of auto-generated URLs, as follows:



The screenshot shows the LunchBadger API Explorer for the 'ConfigObj' model. The table lists the following operations:

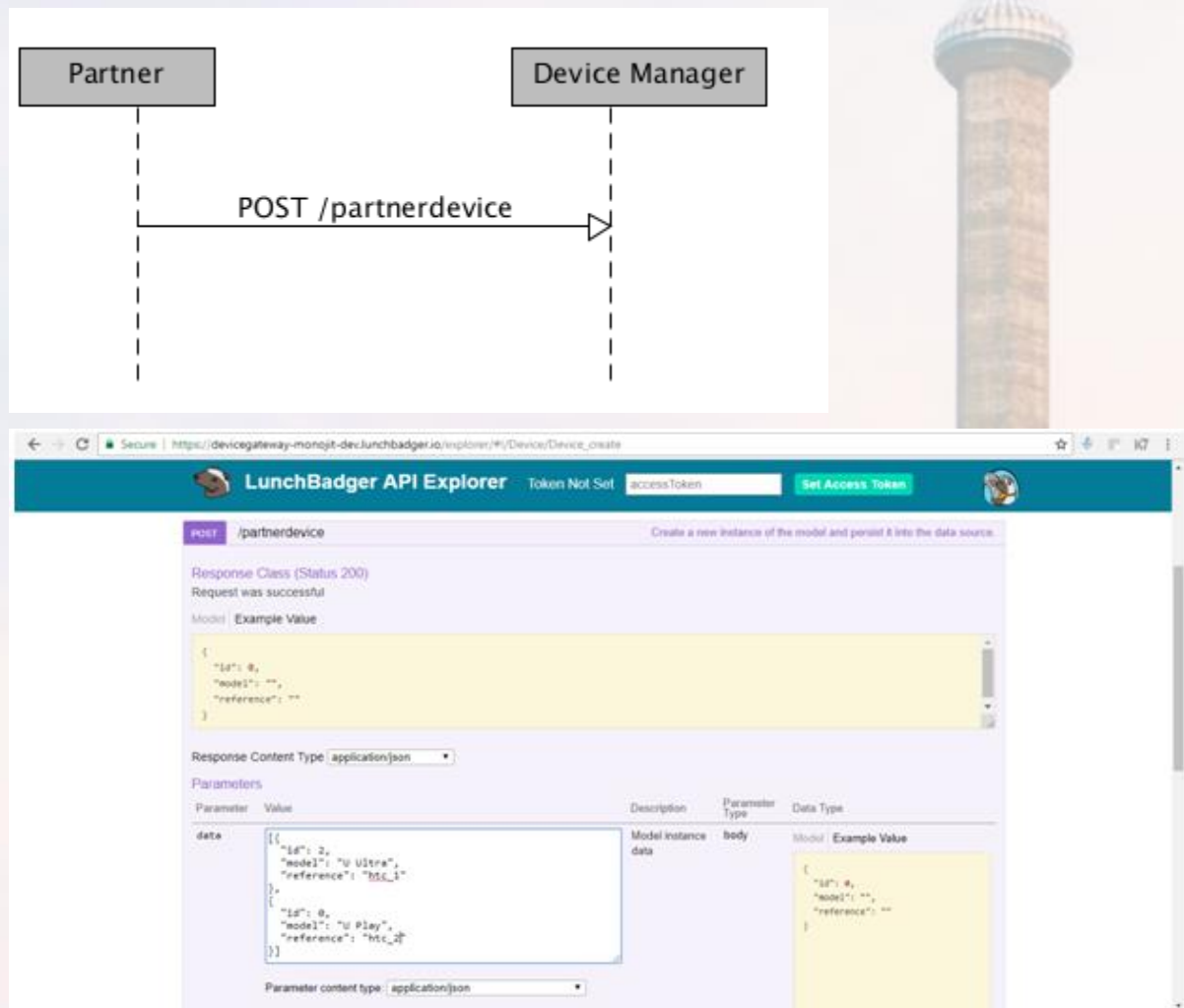
Method	URL	Description
PATCH	/devconfig	Patch an existing model instance or insert a new one into the data source.
GET	/devconfig	Find all instances of the model matched by filter from the data source.
PUT	/devconfig	Patch an existing model instance or insert a new one into the data source.
POST	/devconfig	Creates a new instance of the model and persist it into the data source.
PATCH	/devconfig/{id}	Patch attributes for a model instance and persist it into the data source.
GET	/devconfig/{id}	Find a model instance by {id} from the data source.
HEAD	/devconfig/{id}	Check whether a model instance exists in the data source.
PUT	/devconfig/{id}	Patch attributes for a model instance and persist it into the data source.
DELETE	/devconfig/{id}	Delete a model instance by {id} from the data source.
GET	/devconfig/{id}/exists	Check whether a model instance exists in the data source.
POST	/devconfig/{id}/replace	Replace attributes for a model instance and persist it into the data source.
GET	/devconfig/change-stream	Create a change stream.
POST	/devconfig/change-stream	Create a change stream.
GET	/devconfig/count	Count instances of the model matched by where from the data source.

Functional Use Cases

We just developed the model-based microservices above. The microservices expose model data through REST APIs. But together, they should be able to address the use cases of the overall software system for device provisioning. Let's consider the key use cases.

Use Case 1: If you are a vendor or a partner, you would like to create new configuration objects for your supported device models. This will be a two step process:

- I. Create a new entry for each supported mobile device, using the 'Device' Model, which corresponds to a HTTP POST request:



The diagram illustrates the interaction between a Partner and a Device Manager. A Partner sends a POST request to the Device Manager at the endpoint `/partnerdevice`.

The screenshot shows the LunchBadger API Explorer interface for the `POST /partnerdevice` endpoint. The response class is `Status 200` and the request was successful. The response body is a JSON object:

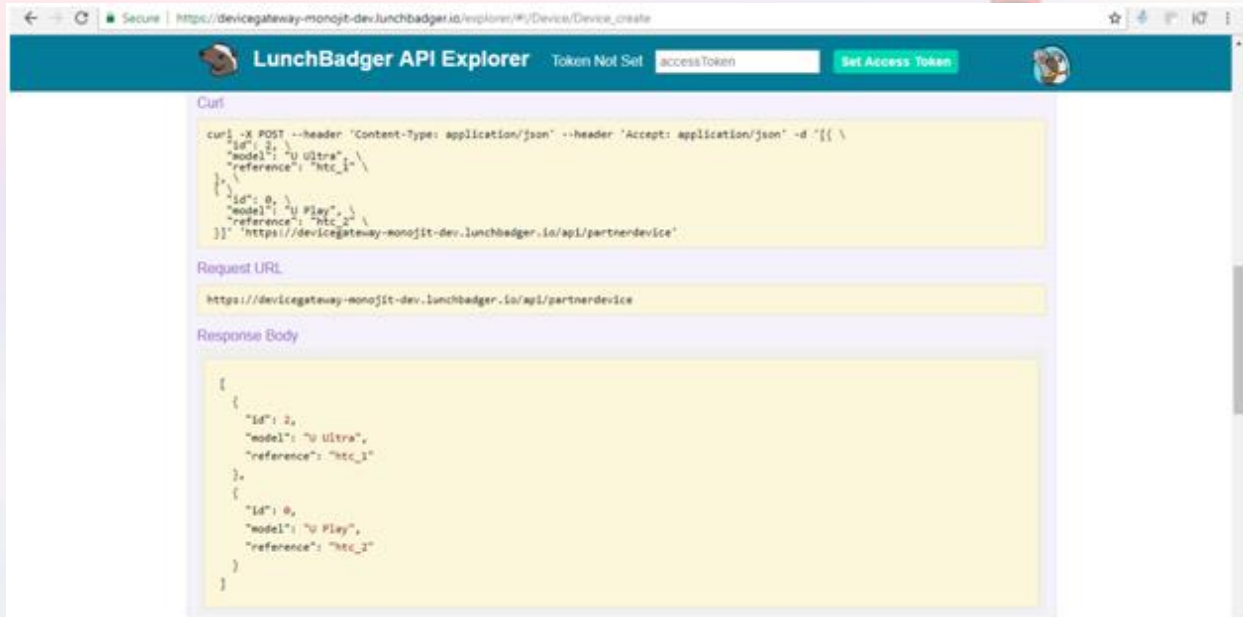
```
{
  "id": 0,
  "model": "",
  "reference": ""
}
```

The parameters section shows a `data` parameter with a value of:

```
{
  "id": 2,
  "model": "U Ultra",
  "reference": "B55_1"
},
{
  "id": 0,
  "model": "U Play",
  "reference": "htc_2"
}
```

The parameter content type is `application/json`.

Tip: Expand the /partnerdevice POST REST API in the API explorer, and fill in a JSON array consisting of device models and corresponding references. Click on the 'Try It Yourself' button to invoke the API. This is like a test environment. Instead you could use curl to invoke the same request.



The screenshot shows the LunchBadger API Explorer interface. The browser address bar displays the URL: `https://devicegateway-monojit-dev.lunchbadger.io/explorer/#/Device/Device_create`. The page title is "LunchBadger API Explorer" and it includes a "Token Not Set" indicator and an "accessToken" input field with a "Set Access Token" button. The main content area is divided into three sections: "Curl", "Request URL", and "Response Body".

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
  "id": 2,
  "model": "U Ultra",
  "reference": "htc_1"
},
{
  "id": 0,
  "model": "U Play",
  "reference": "htc_2"
}]' https://devicegateway-monojit-dev.lunchbadger.io/api/partnerdevice
```

Request URL

```
https://devicegateway-monojit-dev.lunchbadger.io/api/partnerdevice
```

Response Body

```
{
  [
    {
      "id": 2,
      "model": "U Ultra",
      "reference": "htc_1"
    },
    {
      "id": 0,
      "model": "U Play",
      "reference": "htc_2"
    }
  ]
}
```

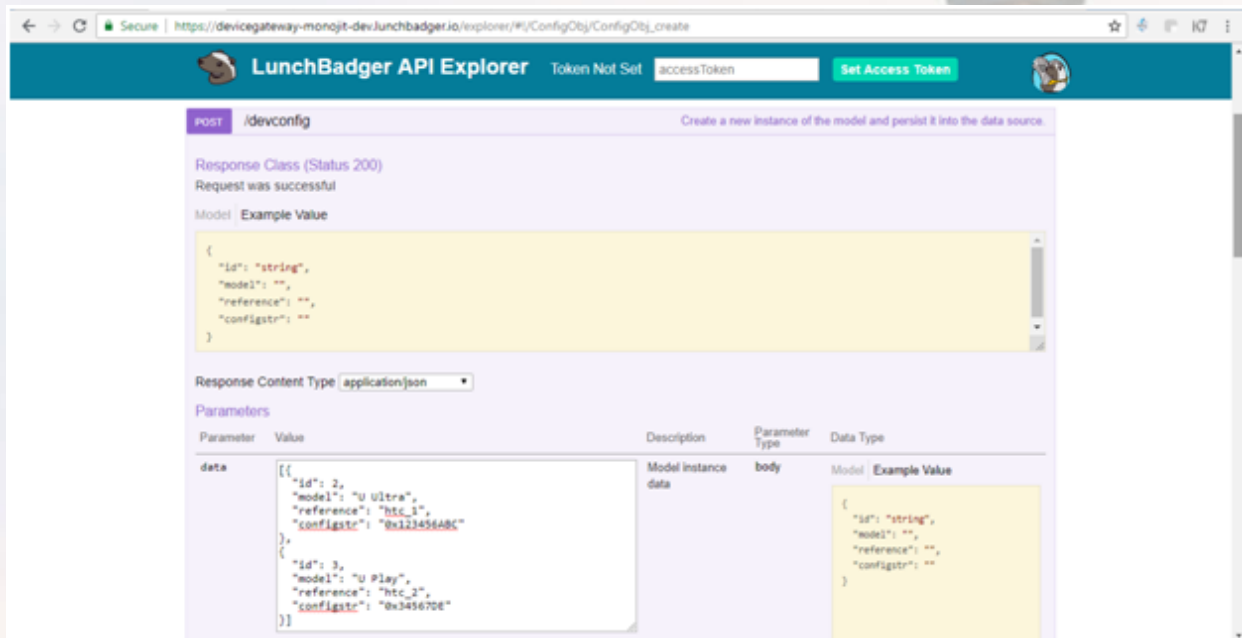
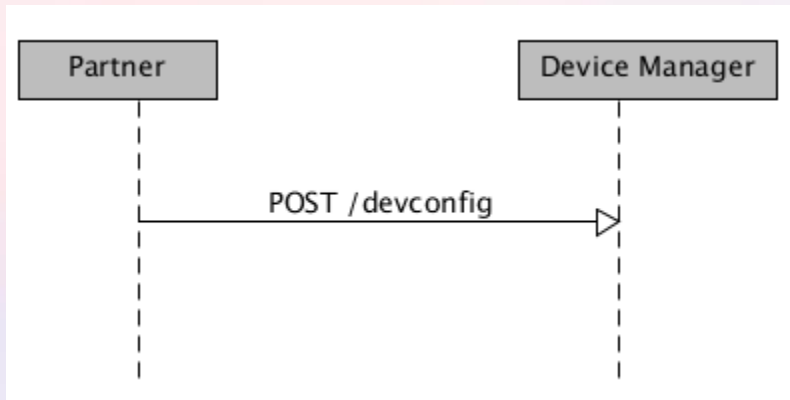
You could also inspect the HTTP Status and Response Headers:



The screenshot shows the HTTP Status and Response Headers section of the API Explorer. The "Response Code" is 200. The "Response Headers" are displayed as a JSON object.

```
{
  "date": "Thu, 13 Sep 2018 19:56:37 GMT",
  "etag": "W/\"5e-YFF7E11h/cvUWViqCvDKKB/\"",
  "x-powered-by": "Express",
  "vary": "Origin",
  "content-type": "application/json; charset=utf-8",
  "access-control-allow-origin": "https://devicegateway-monojit-dev.lunchbadger.io",
  "access-control-allow-credentials": "true",
  "connection": "close",
  "content-length": "94"
}
```

- II. Then, store the corresponding Configuration binaries using the 'ConfigObj' Model:



Secure | https://devicegateway-monojit-dev.lunchbadger.io/explorer/#/ConfigObj/ConfigObj_create

LunchBadger API Explorer | Token Not Set | accessToken | Set Access Token

POST /devconfig | Create a new instance of the model and persist it into the data source.

Response Class (Status 200)
Request was successful

Model Example Value

```
{
  "id": "string",
  "model": "",
  "reference": "",
  "configstr": ""
}
```

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
data	<pre>[{ "id": 2, "model": "U Ultra", "reference": "htc_1", "configstr": "0x123456ABC" }, { "id": 3, "model": "U Play", "reference": "htc_2", "configstr": "0x34567DE" }]</pre>	Model instance data	body	Model Example Value

```
{
  "id": "string",
  "model": "",
  "reference": "",
  "configstr": ""
}
```

Just in case you are interested in verifying that the Data Model is actually connected to a Data Source, you could run queries against MySQL and MongoDB as follows:

```
MariaDB [deviceprovision]> select * from Device where reference like 'htc%';
+-----+-----+-----+
| id | model  | reference |
+-----+-----+-----+
| 0  | U Play | htc_2     |
| 2  | U Ultra| htc_1     |
+-----+-----+-----+
2 rows in set (0.00 sec)

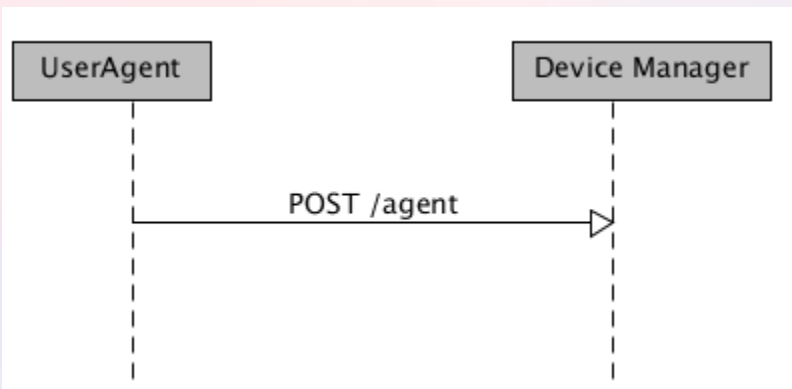
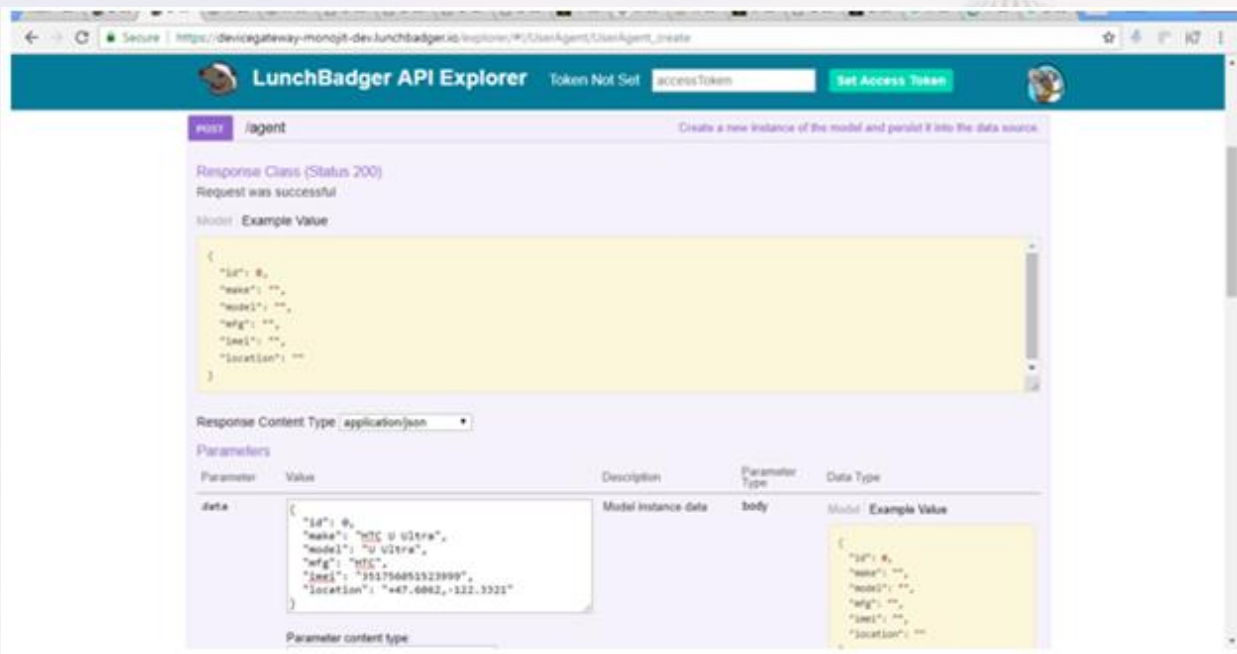
MariaDB [deviceprovision]> █
```

Figure: MySQL Query

```
>
> use deviceconfig
switched to db deviceconfig
>
>
> db.ConfigObj.find({'reference': 'htc_2'})
{ "_id" : 3, "model" : "U Play", "reference" : "htc_2", "configstr" : "@x34567DE" }
>
> █
```

Figure: MongoDB Query

Use Case 2: If you are a device user, you would like to register / enroll your device with the Device Provisioning System, so that new configuration settings can be pushed over the air onto our device. This is again a POST request that is already exposed through the 'UserAgent' microservice.

LunchBadger API Explorer Token Not Set

POST /agent Creates a new instance of the model and persist it into the data source.

Response Class (Status 200)
Request was successful

Model: Example Value

```

{
  "id": 0,
  "make": "",
  "model": "",
  "mfg": "",
  "imei": "",
  "location": ""
}
  
```

Response Content Type: application/json

Parameter	Value	Description	Parameter Type	Data Type
data	<pre> { "id": 0, "make": "HTC U Ultra", "model": "U Ultra", "mfg": "HTC", "imei": "351756051523999", "location": "+37.7749,-122.4194" } </pre>	Model instance data	body	Model: Example Value

Parameter content type

Again, we can verify that the data is consistent with the back-end data source.

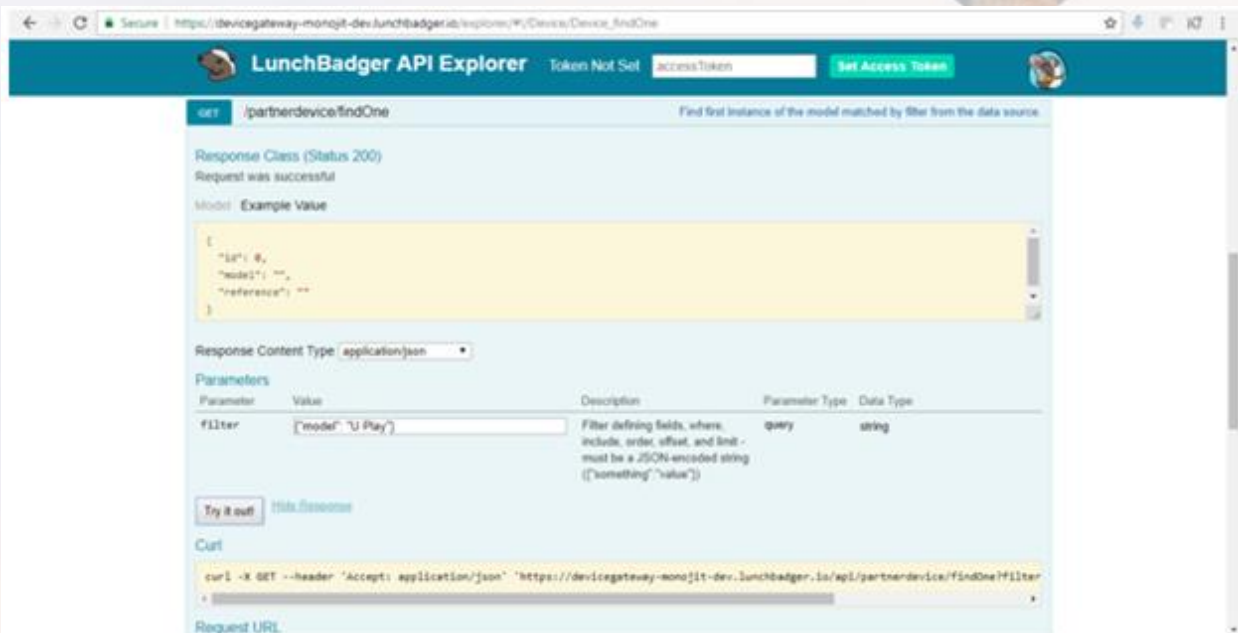
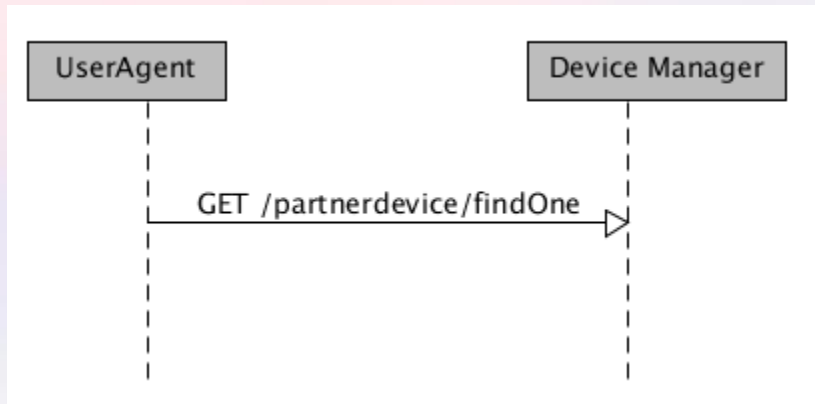
```

MariaDB [deviceprovision]>
MariaDB [deviceprovision]> select * from UserAgent where model = 'U Ultra';
+----+-----+-----+-----+-----+-----+
| id | make  | model | mfg  | imei          | location          |
+----+-----+-----+-----+-----+-----+
| 0  | HTC U Ultra | U Ultra | HTC  | 351756051523999 | +37.7749,-122.4194 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

MariaDB [deviceprovision]>
  
```

Use Case 3: The Device Provisioning System would need APIs to query available configuration settings for a given model. This is again a two step-process:

- I. Find a 'reference' string from the Device Model. For this purpose, a filter (similar to a where clause in SQL) may be used.



GET /partnerdevice/findOne Find first instance of the model matched by filter from the data source.

Response Class (Status 200)
Request was successful
Model: Example Value

```
{
  "id": 4,
  "model": "",
  "reference": ""
}
```

Response Content Type: application/json

Parameter	Value	Description	Parameter Type	Data Type
filter	["model", "U Play"]	Filter defining fields, where, include, order, offset, and limit - must be a JSON-encoded string ({"something":"value"})	query	string

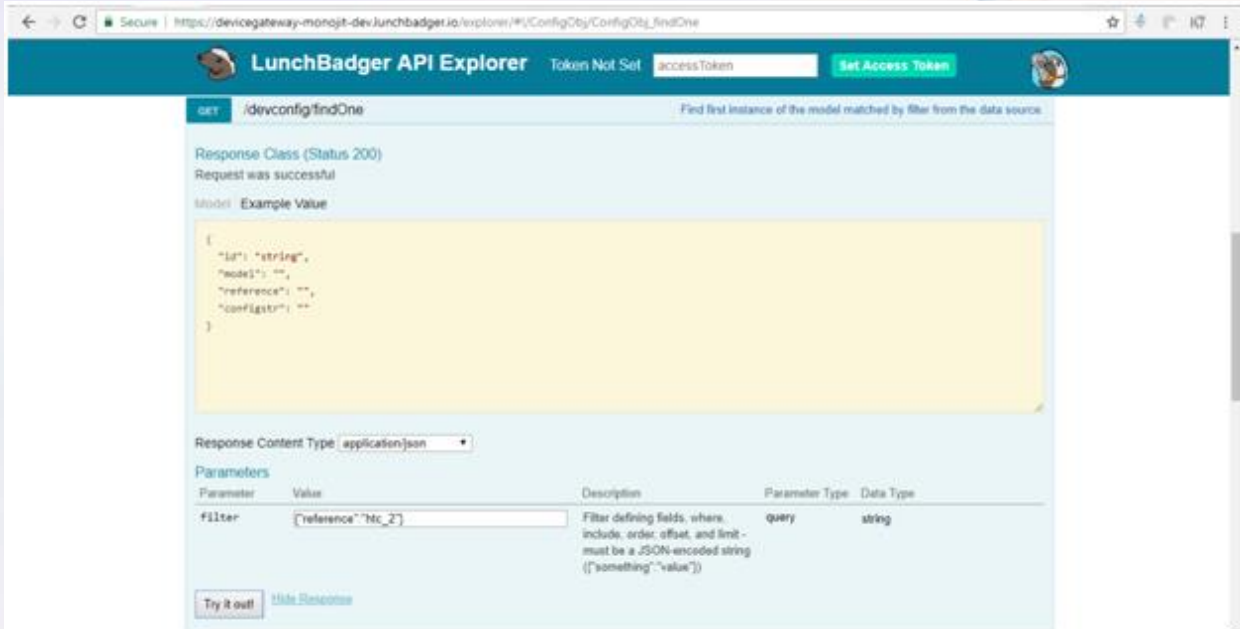
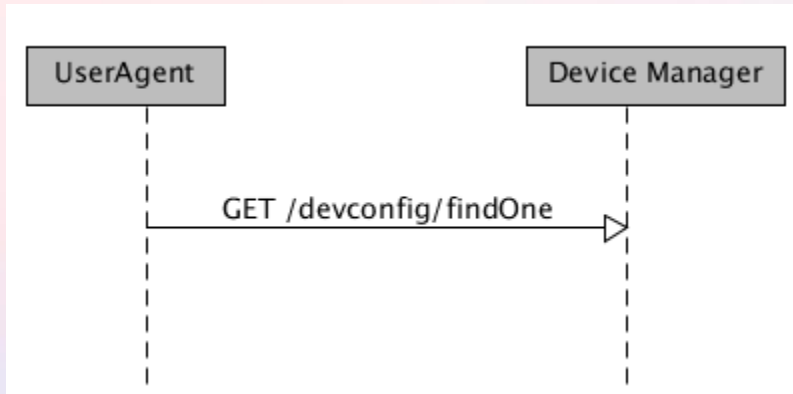
Try it out [View Source](#)

Curl

```
curl -X GET --header 'Accept: application/json' 'https://devicegateway-mono[!t]-dev.lunchbadger.io/api/partnerdevice/findOne?filter=...
```

Request URL

II. Find the configuration object (bytes) from the ConfigObject Model:



GET /devconfig/findOne Find first instance of the model matched by filter from the data source.

Response Class (Status 200)
Request was successful

Model Example Value

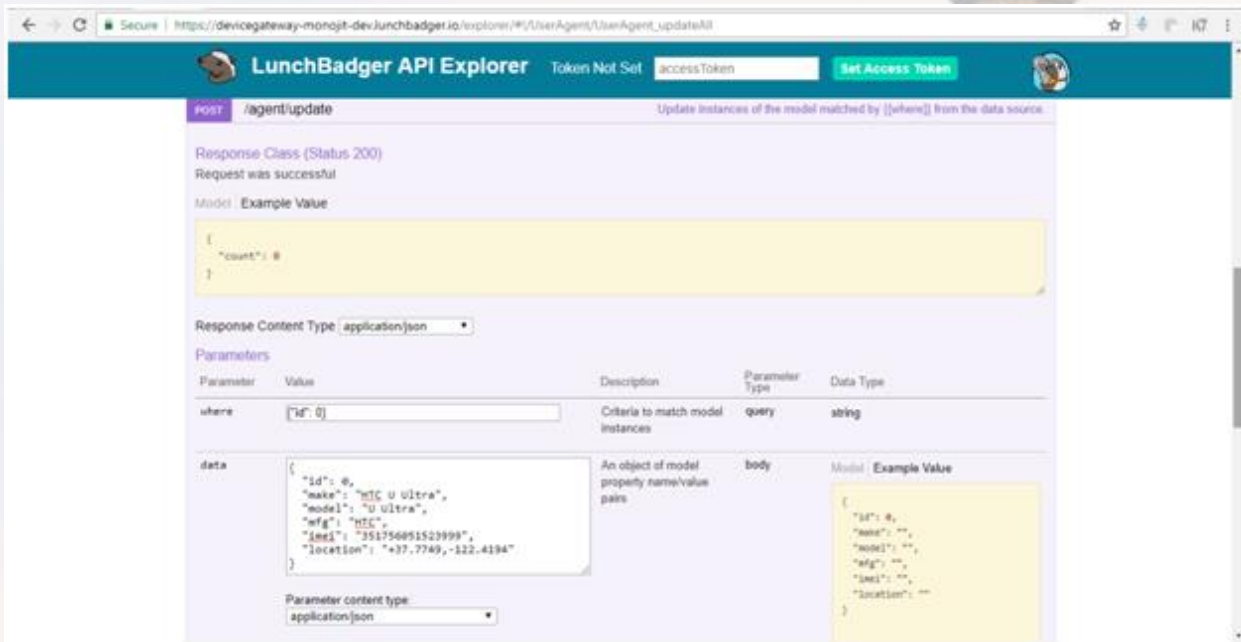
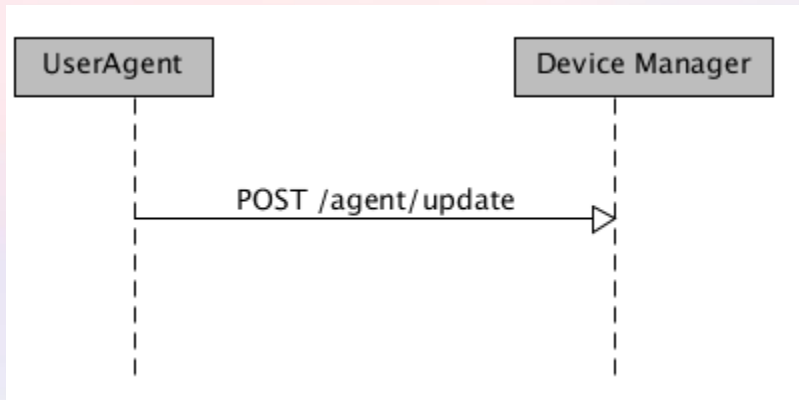
```
{
  "id": "string",
  "model": "",
  "reference": "",
  "configstr": ""
}
```

Response Content Type: application/json

Parameter	Value	Description	Parameter Type	Data Type
filter	<input type="text" value="reference=hc_2"/>	Filter defining fields, where, include, order, offset, and limit - must be a JSON-encoded string ({"something":"value"})	query	string

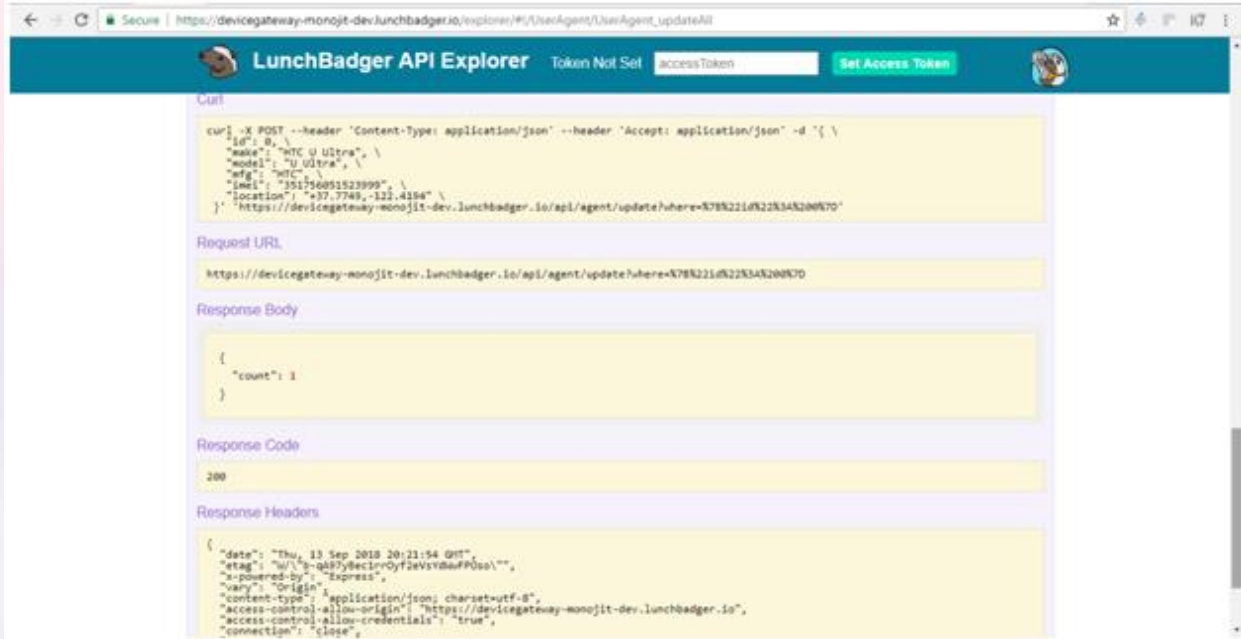
[Try it out!](#) [Hide Response](#)

Use Case 4: Let's just assume the location of a user device (or UserAgent) is of interest to this provisioning system. We may need a way to update the location of a user Device from say Seattle to San Francisco. Here's the corresponding POST request.



The screenshot shows the LunchBadger API Explorer interface for the endpoint `POST /agent/update`. The response class is `Status 200` with the message "Request was successful". The response content type is `application/json`. The response body is a JSON object: `{ "count": 0 }`. The parameters section shows a `where` parameter with the value `["id": 0]` and a description "Criteria to match model instances". The `data` parameter is an object of model property name/value pairs, with an example value: `{ "id": 0, "make": "HTC U Ultra", "model": "U Ultra", "wfg": "HTC", "imei": "351756051520999", "location": "+37.7749,-122.4194" }`. The parameter content type is `application/json`.

Update requests return a response containing the count of the data records updated:



The screenshot shows the LunchBadger API Explorer interface. The request is a POST to `https://devicegateway-monojit-dev.lunchbadger.io/api/agent/update?where=X78N221dN22K34N200N70`. The response body is a JSON object: `{ "count": 1 }`. The response code is 200. The response headers include `date`, `etag`, `x-powered-by`, `x-origin`, `content-type`, `access-control-allow-origin`, `access-control-allow-credentials`, and `connection`.

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{ \
  "id": 0, \
  "name": "HTC U Ultra", \
  "model": "U Ultra", \
  "img": "HTC", \
  "imei": "351796851523999", \
  "location": "+37.7748,-122.4194" \
}' "https://devicegateway-monojit-dev.lunchbadger.io/api/agent/update?where=X78N221dN22K34N200N70"
```

Request URL:

```
https://devicegateway-monojit-dev.lunchbadger.io/api/agent/update?where=X78N221dN22K34N200N70
```

Response Body

```
{ \
  "count": 1 \
}
```

Response Code

```
200
```

Response Headers

```
{ \
  "date": "Thu, 13 Sep 2018 20:11:54 GMT", \
  "etag": "W/\"9-q807y8eclvOyf2eVsVduF0so\"", \
  "x-powered-by": "Express", \
  "x-origin": "Origin", \
  "content-type": "application/json; charset=utf-8", \
  "access-control-allow-origin": "https://devicegateway-monojit-dev.lunchbadger.io", \
  "access-control-allow-credentials": "true", \
  "connection": "close" \
}
```

Summary

In this use case we explained how a fairly complicated real-life application can be built on top of the Express Serverless Platform offered by LunchBadger, in a matter of minutes. We were able to:

- Use off-the-shelf connectors to database back-ends (MySQL and MongoDB) from a palette of connectors available on the Canvas. We showed how this platform supports polyglot persistence
- Create Model Functions, or simply Models, of each entity in our problem domain. Each Data Model will be exposed as a microservice.
- Create an instance of an API Gateway (Express Gateway) and expose REST URLs for accessing the Models. For each model we created a corresponding HTTP request-response pipeline in our API Gateway
- Deployed the application onto a Kubernetes infrastructure, and exposed externally accessible URLs for accessing the application

Next Steps

There are many more nitty-gritty details one would like to take care of when building enterprise applications of this nature. For example:

- Use API keys or OAuth for secure access to APIs
- Configure rate limits for each exposed API
- Add more functionality to the data models, probably through user-defined functions that manipulate the data

The platform offered by LunchBadger addresses these concerns and does lot more.

Sign up for [Free Trial \(no credit card needed\) of Express Serverless Platform](#) and let us know what you think!

END