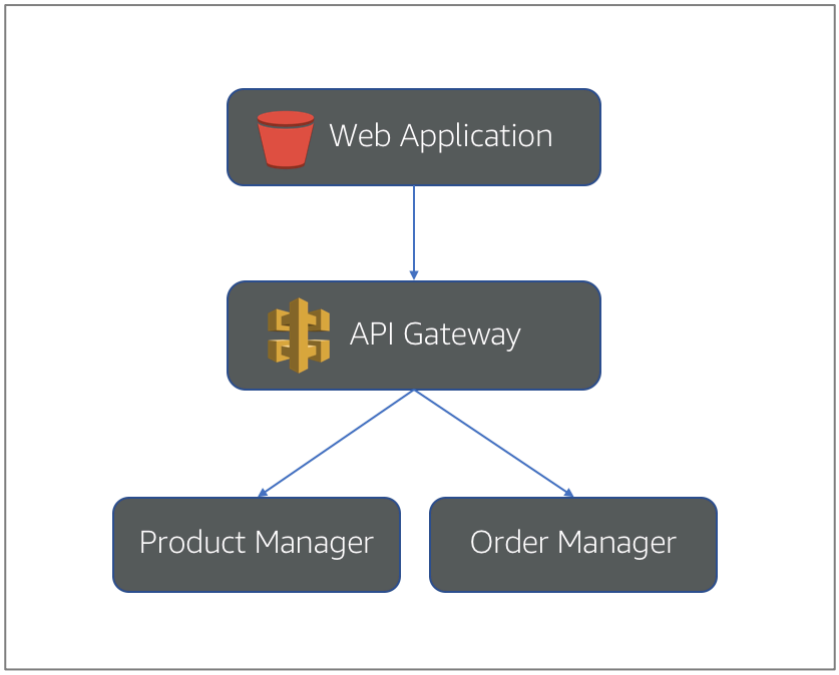
Lab 2 – Building Multi-Tenant Microservices

Overview

In our first lab, we focused all of our attention on getting tenants onboarding and creating a true notion of "SaaS Identity" where user identity was joined to a tenant identity. With those elements in place, we can now turn our attention to thinking about how we actually build the services and functionality of our application in a multi-tenant fashion. This means applying the tenant identity and context with the services that we build.

So far, the services that we've created (tenant registration, user management, etc.) have all been about the fundamentals of onboarding. Now, let's look at the services that we'll introduce to support the actual functionality of our application. For this scenario, we're building a *very* basic order management system. It lets you create a product catalog and place orders against that catalog. It's important to note, that as a multi-tenant SaaS solution the compute and storage resources used to implement this functionality will be shared by all tenants.

Below is a high level diagram of the services that will be delivering this functionality:

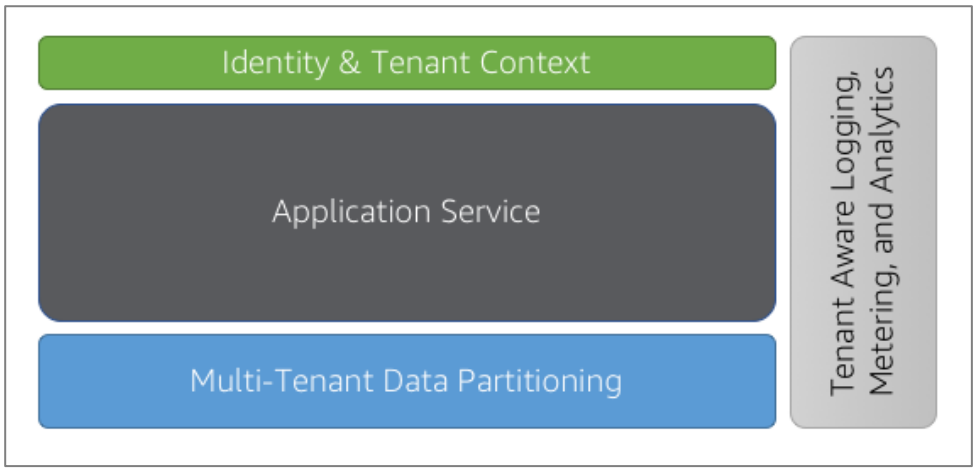


This is a very basic diagram that highlights the services and their interactions with the other aspects of the bootcamp's architecture. Notice that these services are connected to

the web application via the API Gateway, exposing basic CRUD operations to create, read, update, and delete both products and orders.

Of course, as part implementing these services, we must also think about what must be done to apply multi-tenancy to these services. These services will need to store data, log metrics, and acquire user identity all with multi-tenant awareness. So, we have to think about how this is achieved and applied within these services.

Below is a diagram that provides a conceptual view of what it means to build a multi-tenant aware microservice:



This somewhat simplified diagram highlights the key areas we're going to focus on for the multi-tenant microservices we'll be deploying. At the center of the diagram are the actual services being built (in our case the product and order managers). Surrounding it are the areas where we need to factor in multi-tenancy. These include:

- Identity and Tenant Context – our services need some standard way to acquire the current user's role and authorization along with the tenant context. Almost every action within your services happens in the context of a tenant so we'll need to think about how we acquire and apply that context.
- Multi-Tenant Data Partitioning – our two services will need to store data. This means our storage CRUD operations will all need some injection of tenant context to figure out how to partition, persist, and acquire data in a multi-tenant model.
- Tenant Aware Logging, Metering, and Analytics – as we record logs and metrics about activity in our system, we'll need some way to attribute those activities to a specific tenant. Our services must inject this context into any activity messages that are published for troubleshooting or downstream analytics.

This backdrop provides you with a view of the fundamental concepts that we'll explore in this lab. While we won't be writing services from scratch, we'll be highlighting how a service will evolve to incorporate these concepts.

What You'll Be Building

To demonstrate the multi-tenant concepts, we'll go through an evolutionary process where we gradually add incremental multi-tenant awareness to our solution. We'll start with a single-tenant version of our product manager service, then progressively add the bits needed to make this a fully multi-tenant aware service. The basic steps in this process include:

- Deploy a single-tenant product manager microservice – this is a baseline step to illustrate what the service looks like before we begin to layer on the elements of multi-tenancy. It will be a basic CRUD service with no multi-tenant awareness.
- Introduce multi-tenant data partitioning – the first phase of multi-tenancy will be to add the ability to partition data based on a tenant identifier supplied as part of an incoming request.
- Extract tenant context from user identity – add the ability to use the security context of HTTP calls into the service to extract and apply tenant identity for our data partitioning scheme.
- Introduce a second tenant to demonstrate partitioning – register a new tenant and manage products through that tenant context to illustrate how the system has successfully partitioned the data in the application.
- Log data with tenant context – in the last step we'll demo how tenant context should also be injected into logs and metering to support tenant-centric analytics.

When this process is finally completed, you should have a complete multi-tenant aware product manager service that supports both data partitioning, extraction of tenant context from a user's identity, and the ability to inject tenant identity into logs.

Part 1 – Deploying a Single-Tenant Product Manager Service

Before we can see how multi-tenancy influences the business services of our application, we need to see a baseline single-tenant service in action. This will provide a foundation for our exploration of multi-tenancy, illustrating how multi-tenancy influences the implementation of our microservice.

In this basic model, we'll deploy a service, create a DynamoDB table to hold our product data, then use cURL commands to exercise this new service. This solution will also setup an API Gateway experience, but we'll let the provisioning scripts configure this aspect of the solution (since you've already gotten a taste of configuring resources and methods in the prior lab).

Step 1 – Let's start this process by taking a closer look at the single-tenant service that we'll be deploying. Like the prior services implemented in the first lab, the product manager microservice is built with Node.js and Express. It exposes a series of CRUD operations via a REST interface.

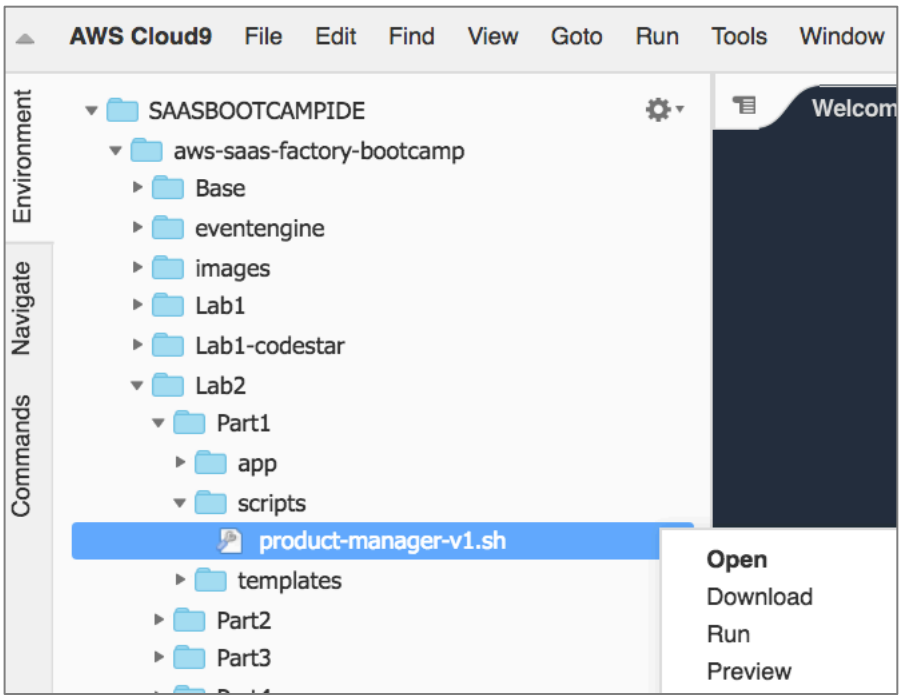
The source code for this file is available at Lab2/Part1/app/source/product-manager/src/server.js.

In looking at this file, you'll see a number of entry points that correspond to the REST methods (app.get(…), app.post(…), etc.). Within each of these functions is the implementation of the corresponding REST operation. Let's take a look at one of these methods in more detail to get a sense of what's going on here.

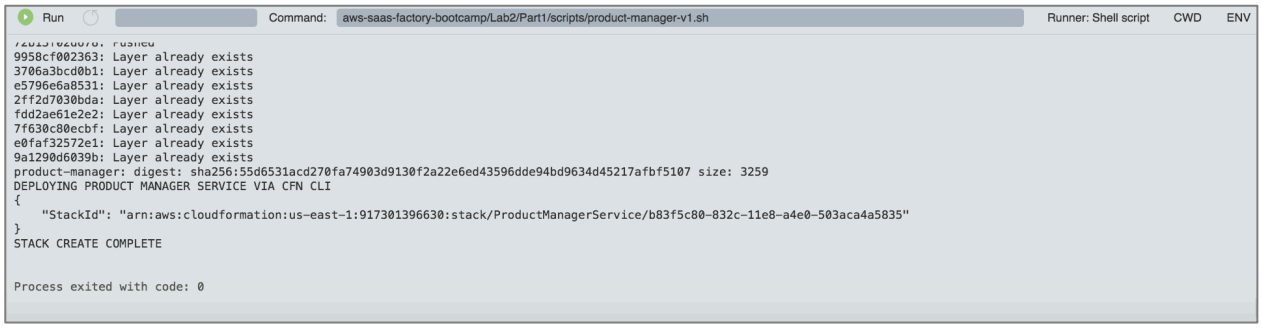
```
51  app.get('/product/:id', function (req, res) {
52      winston.debug('Fetching product: ' + req.params.id);
53      // init params structure with request params
54      var params = {
55          productId: req.params.id
56      };
57      tokenManager.getSystemCredentials(function (credentials) {
58          // construct the helper object
59          var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
60          dynamoHelper.getItem(params, credentials, function (err, product) {
61              if (err) {
62                  winston.error('Error getting product: ' + err.message);
63                  res.status(400).send('{"Error" : "Error getting product"}');
64              }
65              else {
66                  winston.debug('Product ' + req.params.id + ' retrieved');
67                  res.status(200).send(product);
68              }
69          });
70      });
71  });
```

Let's start by looking at the signature of the method on line 51. Here you'll see the traditional REST path for a **GET** method with the "*/product*" resource followed by an identifier parameter that indicates which product is to be fetched. This value is extracted from the incoming request on line **55** and populated into a "params" structure. The rest of this function is about calling a DynamoDBHelper to fetch the item from a DynamoDB table.

Step 2 – Our first step is to deploy the single-tenant product manager, within our Cloud9 IDE. Navigate to **Lab2/Part1/scripts** directory, right-click **product-manager-v1.sh**, and click Run to execute the shell script.



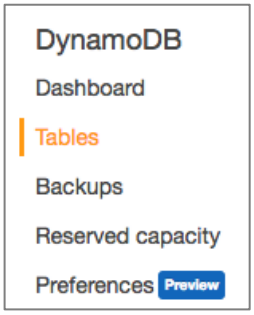
Step 3 – Wait for the **product-manager-v1.sh** shell script to execute successfully, as confirmed by the "**Process exited with code: 0**" message, and the confirmation



message "**STACK CREATE COMPLETE**".

Step 4 – Now that our service is deployed, we have to introduce its supporting elements. Let's start by creating the **DynamoDB** table that will be used to store product information. First, navigate to the DynamoDB service in the AWS console.

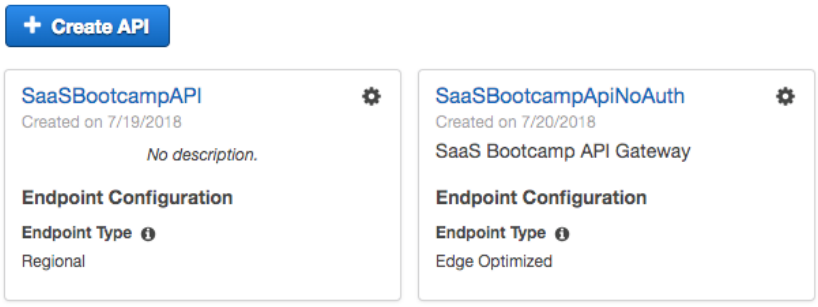
Step 5 – Once you're in the **DynamoDB** console, select the "Table" option from the navigation list in the upper left-hand side of the page (show here).



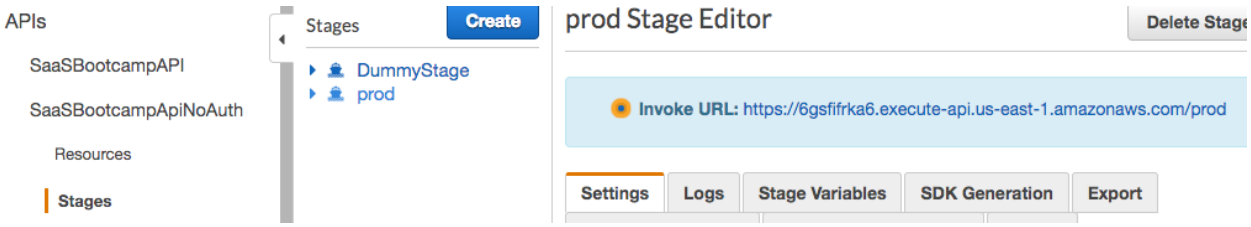
Step 6 – Now choose the "Create Table" button at the top of the page to start the table creation process.

Step 7 – You'll now need to enter the attributes of your new table. Enter a table name of "ProductBootcamp" and a primary key of "productId". Once you've entered this data, select the "Create" button at the bottom right of the page to launch the table creation process.

Step 8 – At this point the table should be created and your service should be running and accessible via the Amazon API Gateway. First, we need the API Gateway URL that exposes our microservices. Navigate to API Gateway in the AWS console and select the **SaaSBootcampApiNoAuth** API. This API was created automatically for you at the end of Lab 1 and includes all resources and methods required to run our SaaS application.



Step 9 – Select "Stages" from the left-hand menu and then click on the "prod" stage. The invocation URL will be displayed. This is the base URL (including /prod at the end) that all of your microservices will be accessible from.



Step 10 – Now let's verify that the basic plumbing of our Product Manager service is all in place by making a simple call to its health check endpoint. For this step and subsequent steps, you can use either cURL or Postman (or the tool of your choice). Let's invoke the GET method on "/product/health" to verify that the service is running.

```
curl http://API-GATEWAY-PROD-URL/product/health
```

You should get an HTTP 200 response with a JSON formatted success message from the cURL command indicating that the request was successfully processed and the service is ready to process requests.

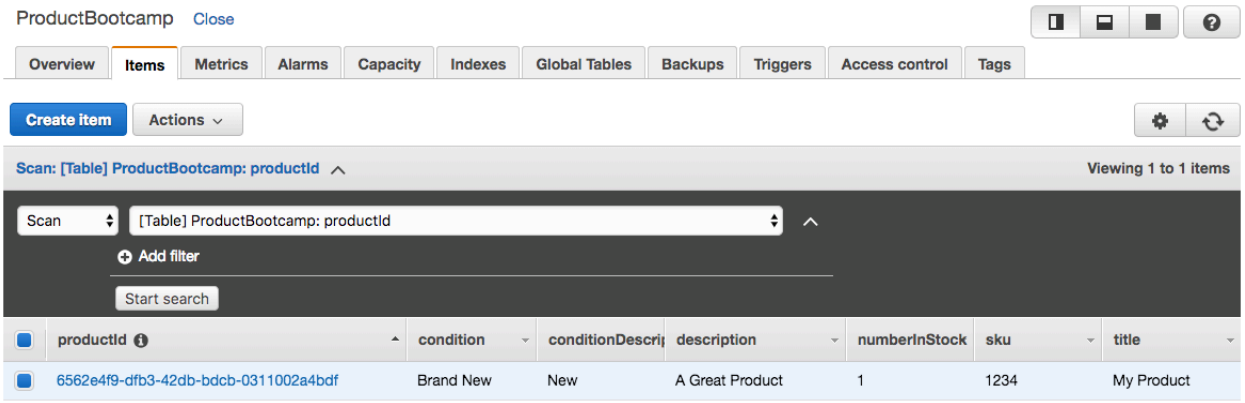
Step 11 – Now that we know the service is up-and-running, we can add a new product to the catalog via the REST API. Submit the following REST command to create your first product:

```
curl --header "Content-Type: application/json" --request POST --data
'{"sku": "1234", "title": "My Product", "description": "A Great
Product", "condition": "Brand New", "conditionDescription": "New",
"numberInStock": "1"}' https://API-GATEWAY-PROD-URL/product
```

Step 12 – Let's now go verify that the data we submitted landed successfully in the DynamoDB table we created. Navigate to the DynamoDB service in the AWS console and select "Tables" from the list of options at the upper left-hand side of the page.

Step 13 – The center of the page should now display a list of tables. Find your "ProductBootcamp" table and select the link with the table name. This will display basic information about the table.

Step 14 – Select the "Items" tab from the top of the screen, you'll see the list of items in your product table, which should include the item you just added.



Recap: This initial exercise illustrates a single-tenant version of the product manager service. It does not have identity or tenant context built into the service. In many respects, this represents the flavor of service you'd see in many non-SaaS environments. It gives us a good base for thinking about how we can now evolve the service to incorporate multi-tenant concepts.

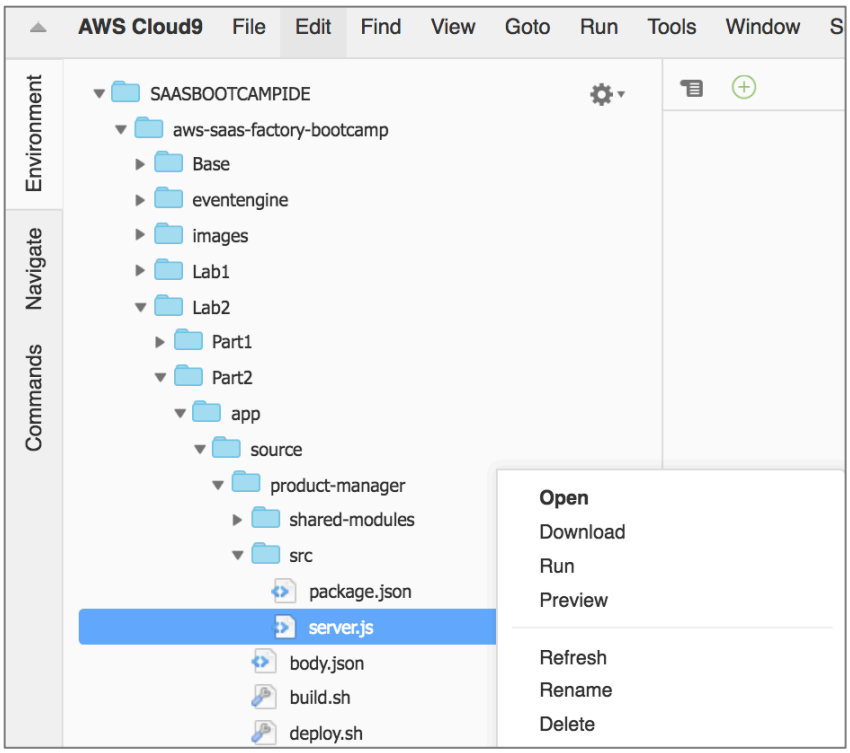
Part 2 – Adding Multi-Tenant Data Partitioning

The first step in making our service multi-tenant aware is to implement a partitioning model where we can persist data from multiple tenants in our single DynamoDB database. We'll also need to inject tenant context into our REST requests and leverage this context for each of our CRUD operations.

To make this work, will also need a different configuration for our DynamoDB database, introducing a tenant identifier as the partition key. We'll also need a new version of our service that accepts a tenant identifier in each of the REST methods and applies it as it accesses DynamoDB tables.

The steps that follow will take you through the process of adding these capabilities to the product manager service:

Step 1 – For this iteration, we'll need a new version of our service. While we won't modify the code directly, we'll take a quick look at how the code changes to support data partitioning. Open our product manager **server.js** file in our Cloud9 IDE. In Cloud9, navigate to "**Lab2/Part2/app/source/product-manager/**", right-click **server.js** and click "Open."



This file doesn't look all that different than our prior version. In fact, the only change here is that we've added a tenant identifier to the parameters that we supply to the DynamoDBHelper. Below is a snippet of the code from our file.

```
112 app.post('/product', function(req, res) {
113     var product = req.body;
114     product.productId = uuidV4();
115     product.tenantId = req.body.tenantId;
116
117     // construct the helper object
118     tokenManager.getSystemCredentials(function (credentials) {
119         var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
120         dynamoHelper.putItem(product, credentials, function (err, product) {
121             if (err) {
122                 winston.error('Error creating new product: ' + err.message);
123                 res.status(400).send('{"Error" : "Error creating product"}');
124             }
125             else {
126                 winston.debug('Product ' + req.body.title + ' created');
127                 res.status(200).send({status: 'success'});
128             }
129         });
130     });
131 });
```

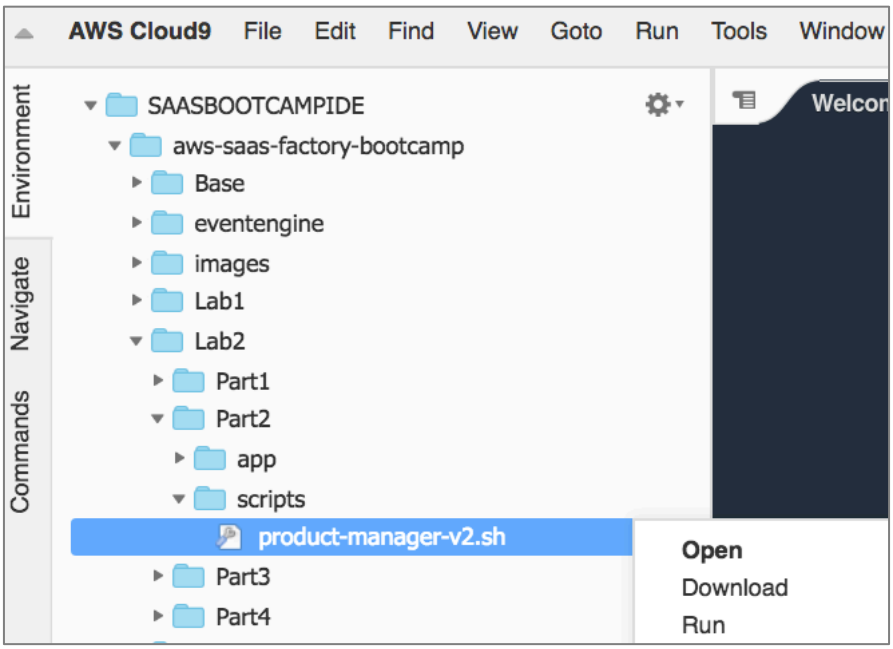
Line 115 represents the only change you'll see on the surface here. It extracts the tenant identifier from the incoming request and adds it to our product object. This, of course, means that REST calls to this method must also supply a tenant identifier with each invocation of this method.

Step 2 – Up to this point, we haven't really looked at the **DynamoDBHelper** to see how it accommodates our attempts to get items from the DynamoDB table. In most respects, this module is wrapper of the AWS provided DynamoDB client, adding elements to support isolation. In fact, even as we're introducing this tenant identifier model, it does not change how **DynamoDBHelper** processes this request. Below is a snipped of code from the **DynamoDBHelper getItem()** method:

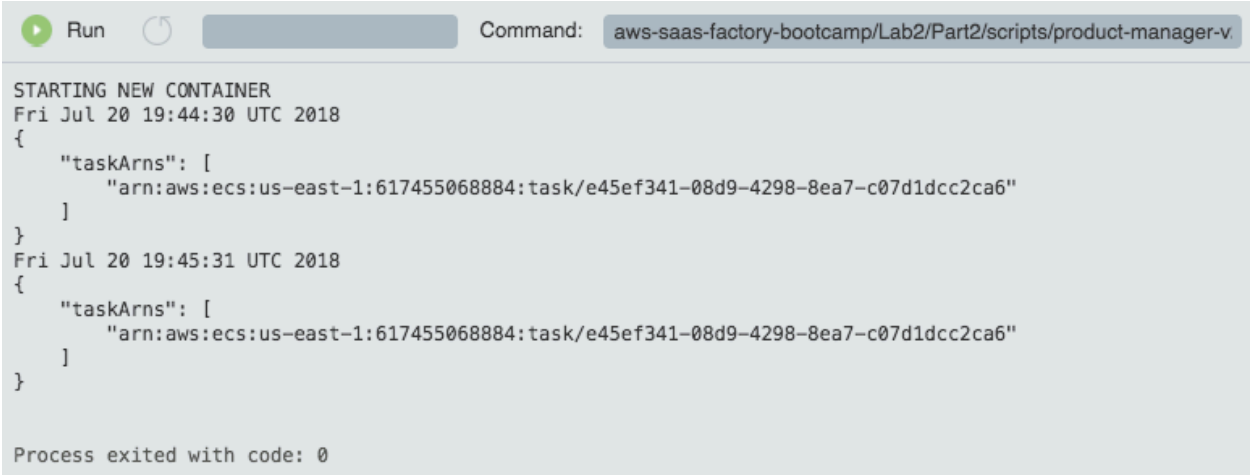
```
100 DynamoDBHelper.prototype.getItem = function(keyParams, credentials, callback) {
101     this.getDynamoDBDocumentClient(credentials, function (error, docClient) {
102         var fetchParams = {
103             TableName: this.tableDefinition.TableName,
104             Key: keyParams
105         };
106
107         docClient.get(fetchParams, function(err, data) {
108             if (err)
109                 callback(err);
110             else
111                 callback(null, data.Item);
112         });
113     }.bind(this));
114 };
```

You'll notice that on line 104 we're passing through the parameters that we constructed in our product manager service. The client will simply use the parameters to match the **partition key** for the table. The key takeaway here is that nothing unique is done in the code to support the partitioning by a tenant identifier. It's simply just another key in our DynamoDB table.

Step 3 – Now that you have a better sense of how this service changes to accommodate data partitioning, let's go ahead and deploy version 2 of the product manager, within our Cloud9 IDE. Navigate to **Lab2/Part2/scripts** directory, right-click **product-manager-v2.sh**, and click Run to execute the shell script.

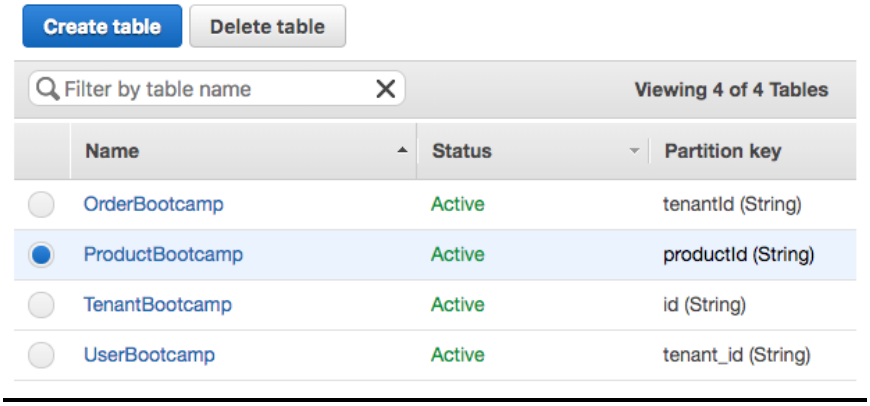


Step 4 – Wait for the **product-manager-v2.sh** shell script to execute successfully, as confirmed by the "**Process exited with code: 0**" message.



Step 5 – With this new partitioning scheme, we must also change the configuration of our DynamoDB table. If you recall, the current table used "productId" as the partition key. We now need to have "tenantId" be our partition key and have the product id serve as a secondary index (since we may still want to sort on that value). The easiest way to introduce this change is to simply delete the existing product table and create a new one with the correct configuration. To start this process, navigate to the DynamoDB service in the AWS console and select the "Tables" option from the menu at the top left of the page.

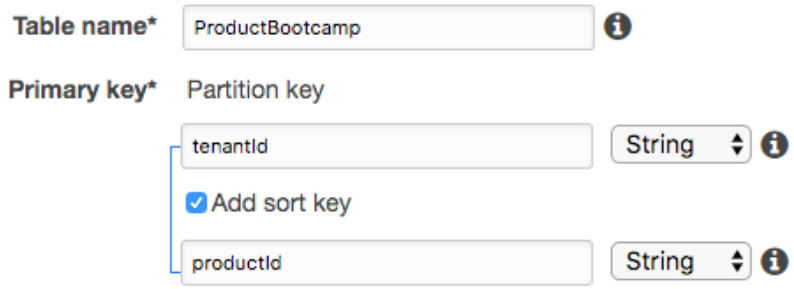
Step 6 – Now select the radio button for the "ProductBootcamp" table (the screen below provides an example of this part of the page).



Step 7 – After selecting the product table, select the "Delete Table" button to start the deletion of the table. You'll be prompted to confirm removal of CloudWatch logs to complete the process.

Step 8 – Now we can start the table creation process from scratch. Select the "Create Table" button from the top of the page.

Step 9 – You're must now enter the attribute for this new version of the product table. Enter "**ProductBootcamp**" for the table name, "**tenantId**" for the partition key, then click on the "**Add sort key checkbox**" and we'll move "**productId**" to the sort key.



Step 10 – The service has now been modified to support the introduction of a tenant identifier and we've modified DynamoDB to partition the data with this tenant identifier. It's time now to validate that the new version of the service is up-and-running. Issue the following cURL command to invoke the health check on the service. Refer to Part 1 if you need to find your API Gateway URL.

```
curl http://API-GATEWAY-PROD-URL/product/health
```

Upon invoking this command, you should get a 200 response from your request indicating that the request was successfully processed and the service is ready to process requests.

Step 11 – Now that we know the service is up-and-running, we can add a new product to the catalog via the REST API. Unlike our prior REST call, this one must provide the tenant identifier as part of the request. Submit the following REST command to create a product for tenant "123":

```
curl --header "Content-Type: application/json" --request POST --data
'{"tenantId": "123", "sku": "1234", "title": "My Product",
"description": "A Great Product", "condition": "Brand New",
"conditionDescription": "New", "numberInStock": "1"}' https://API-
GATEWAY-PROD-URL/product
```

This looks much like the prior example. However, notice that we pass a parameter of tenant id ("123") in the body.

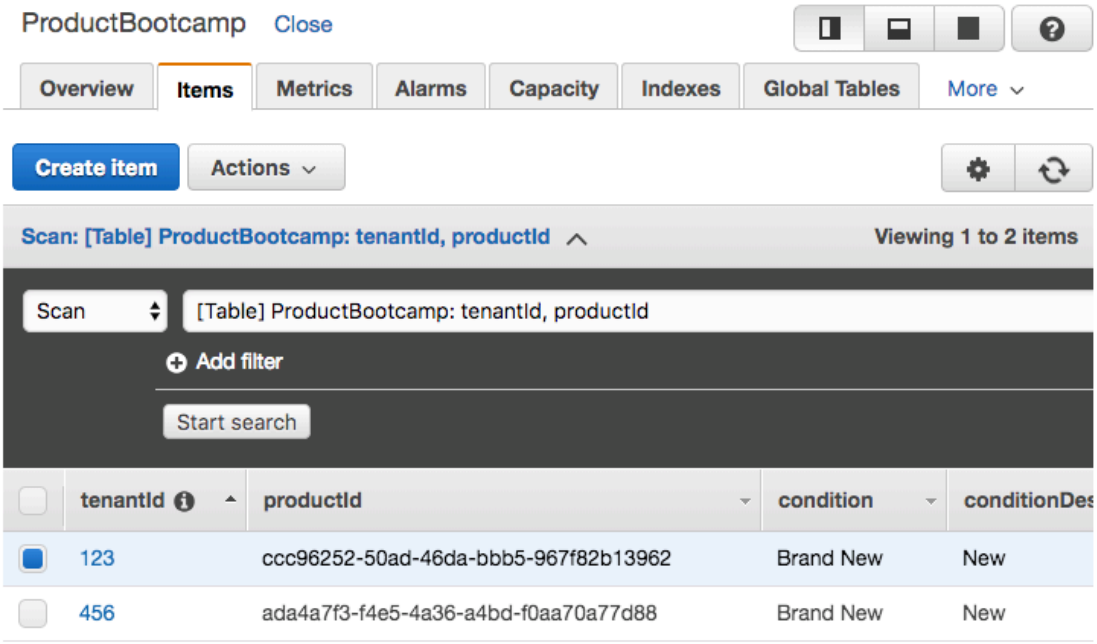
Step 12 – Before we verify that this data was successfully written, let's introduce another product for a different tenant. This will highlight the fact that our partitioning scheme can store data separately for each tenant. To add another product for a different tenant, we just issue another POST command for a different tenant. Submit the following POST for tenant "456".

```
curl --header "Content-Type: application/json" --request POST --data
'{"tenantId": "456", "sku": "1234", "title": "My Product",
"description": "A Great Product", "condition": "Brand New",
"conditionDescription": "New", "numberInStock": "1"}' https://API-
GATEWAY-PROD-URL/product
```

Step 13 – Assuming your request succeeded, let's now go verify that the data we submitted landed successfully in the DynamoDB table we created. Navigate to the DynamoDB service in the AWS console and select "Tables" from the list of options at the upper left-hand side of the page.

Step 14 – The center of the page should now display a list of tables. Find your "product" table and select the link with the table name. This will display basic information about the table.

Step 15 – Finally, if you select the "Items" tab from the top of the screen, you'll see the list of items in your product table, which should include the two items you just added. Verify that these two items have are partitioned based on the two tenant identifiers that you suppled ("123" and "456").



Recap: You've now successfully introduced data partitioning into your service. The microservice achieved this by adding a tenant identifier parameter to the product manager service and changing the product table to use "**tenantId**" as the partition key. Now, all of your CRUD operations are multi-tenant aware.

Part 3 – Acquiring and Applying Tenant Context

At this stage we have data partitioning, but our way of introducing the tenant context was somewhat crude. It's simply not practical or secure to expect that tenant identifiers are to be passed as a parameter to every call to the product management service. Instead, this tenant context should come from the tokens that are part of the authentication process that we setup in the prior lab.

So, our next step is to enable our service to be aware of these security tokens and extract our tenant context from these tokens. Then, our partitioning that we just setup can rely on a tenant identifier that was provisioned during onboarding and simply flowed through to your product manager service in the header of each HTTP request.

For this section, we'll see how our product manager service gets retrofitted with new code to extract these tokens from the HTTP request and applies them to our security and data partitioning model.

The steps that follow take you through the process of deploying a V3 of our product manager service that includes these capabilities:

Step 1 – For this iteration, we'll need a new version of our service. While we won't modify the code directly, we'll take a quick look at how the code changes to support acquiring tenant context from identity tokens. View the new version of

our Product Manager service in Cloud9 by opening Lab2/Part3/app/source/product-manager/src/server.js

The file at this location introduce a new TokenManager helper object that abstracts away many aspects of the token processing. Let's take a look at a snippet of the V3 version to see how tenant context is acquired from the user's identity:

```
34  app.use(function(req, res, next) {
35      res.header("Access-Control-Allow-Origin", "*");
36      res.header("Access-Control-Allow-Methods", "GET, POST, OPTIONS, PUT, PATCH, DELETE");
37      res.header("Access-Control-Allow-Headers", "Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With");
38      bearerToken = req.get('Authorization');
39      if (bearerToken) {
40          tenantId = tokenManager.getTenantId(req);
41      }
42      next();
43  });
```

The getTenantId function shown here, which appears in most of the services of our application, provides the mechanism for acquiring the tenant identifier from the HTTP header for each REST request. It achieves this by using the middleware construct of the Express framework. This middleware allows you to introduce a function that intercepts and pre-processes each request before it is processed by the functions for each REST method.

In looking at this code, you'll see that line 38 extracts the bearer token from the HTTP authorization header. This token holds the data we want to use to acquire our tenant context. Then, on line 40, the code uses the TokenManager helper to get the tenant identifier out of the request. The call to this function returns the tenant identifier and assigns it to the tenantId variable. This variable is then referenced throughout the service to acquire tenant context.

Step 2 – Now that you have the tenant identifier, the application of this is relatively straightforward. Here, we'll change the way we're acquiring the tenant identifier, referencing the tenantId that we populated in Step 1 (instead of pulling this from the incoming requests).

```
67  app.get('/product/:id', function(req, res) {
68      winston.debug('Fetching product: ' + req.params.id);
69       // init params structure with request params
70      var params = {
71          tenantId: tenantId,
72          productId: req.params.id
73      };
74
75      tokenManager.getSystemCredentials(function (credentials) {
76          // construct the helper object
77          var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
78          dynamoHelper.getItem(params, credentials, function (err, product) {
79              if (err) {
80                  winston.error('Error getting product: ' + err.message);
81                  res.status(400).send('{"Error" : "Error getting product"}');
82              }
83              else {
84                  winston.debug('Product ' + req.params.id + ' retrieved');
85                  res.status(200).send(product);
86              }
87          });
88      });
89  });
```

In looking at line 71, you'll see that we have changed this line to reference the tenantId that we extracted from the token in the middleware.

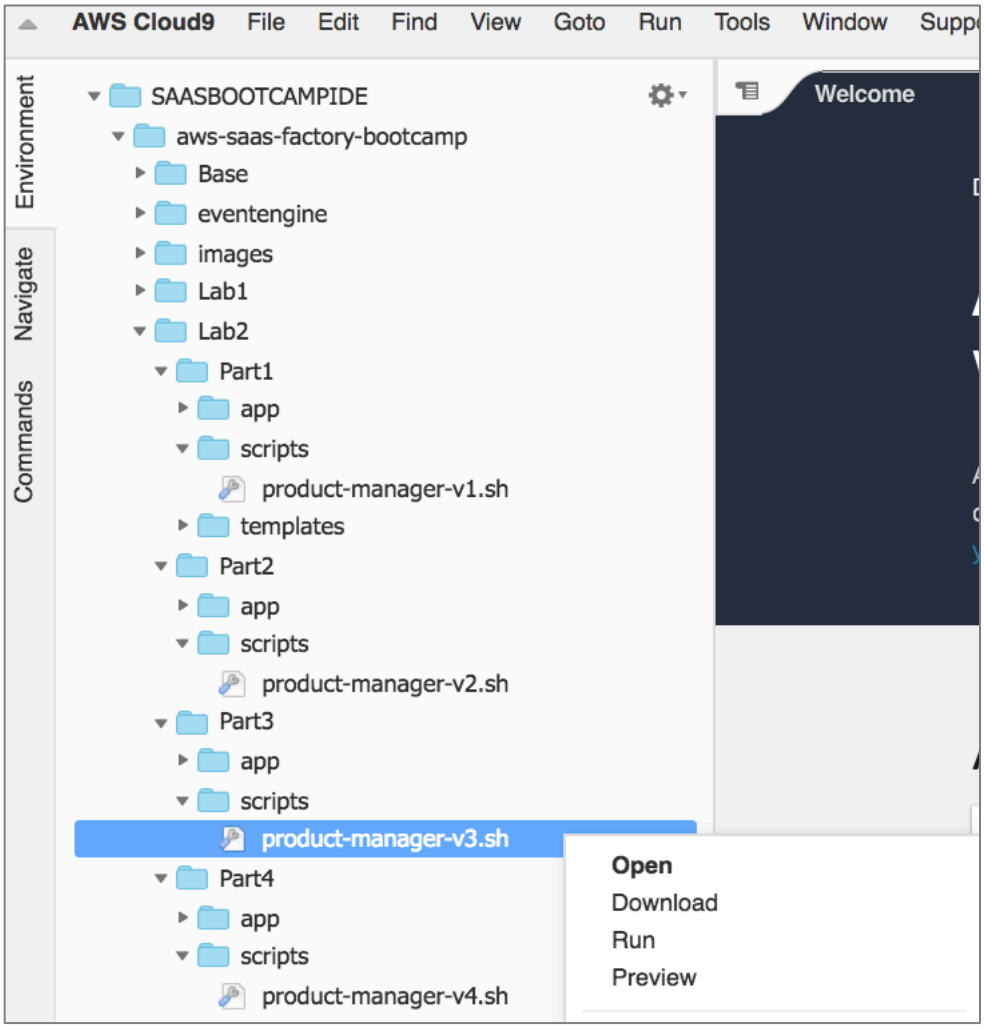
Step 3 – As you can imagine, most of the token processing work here is intentionally embedded in the helper class. Let's take a quick look at what is in that class to see how it's extracting this context from the tokens.

This function extracts the security token from the Authorization header of the HTTP request, decodes it, then acquires the tenantId from the decoded token (on line 33). In a production environment, this unpacking would use a signed certificate to create a more secure model for unpacking tokens. Below is a snippet of the code from the TokenManager that was invoked to extract the token.

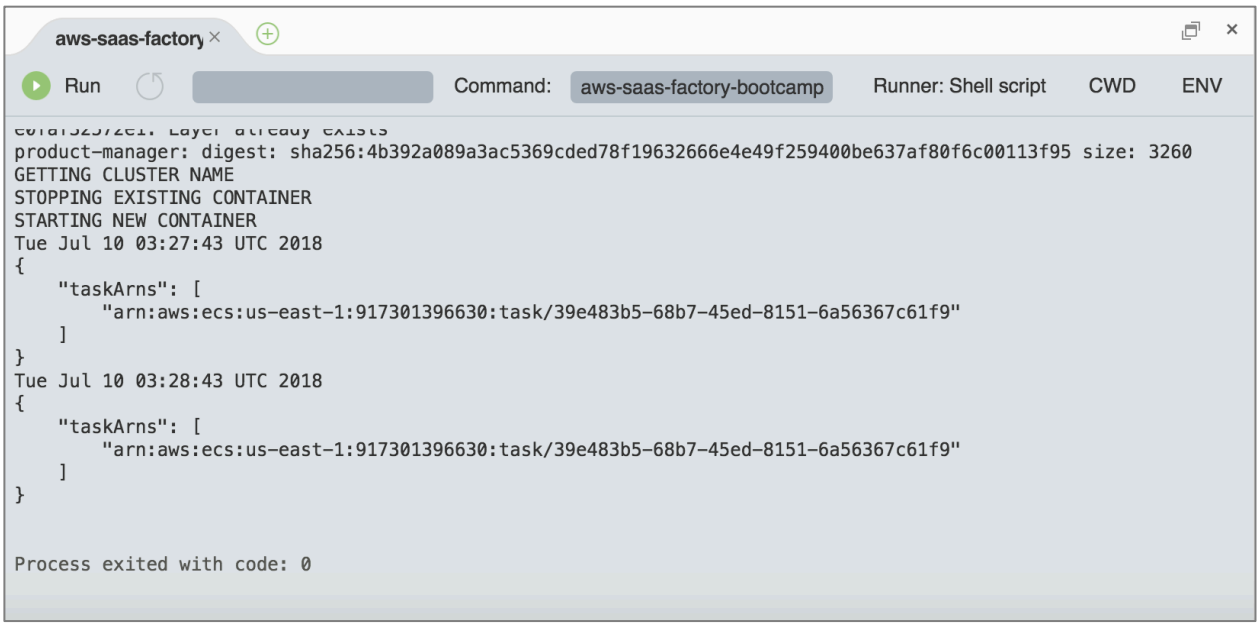
```
26  module.exports.getTenantId = function(req) {
27      var tenantId = '';
28      var bearerToken = req.get('Authorization');
29      if (bearerToken) {
30          bearerToken = bearerToken.substring(bearerToken.indexOf(' ') + 1);
31          var decodedIdToken = jwtDecode(bearerToken);
32          if (decodedIdToken)
33              tenantId = decodedIdToken['custom:tenant_id'];
34      }
35      return tenantId;
36  };
```

Step 4 – Now that you have a better sense of how we've introduced tenant context through HTTP headers, let's go ahead and deploy it.

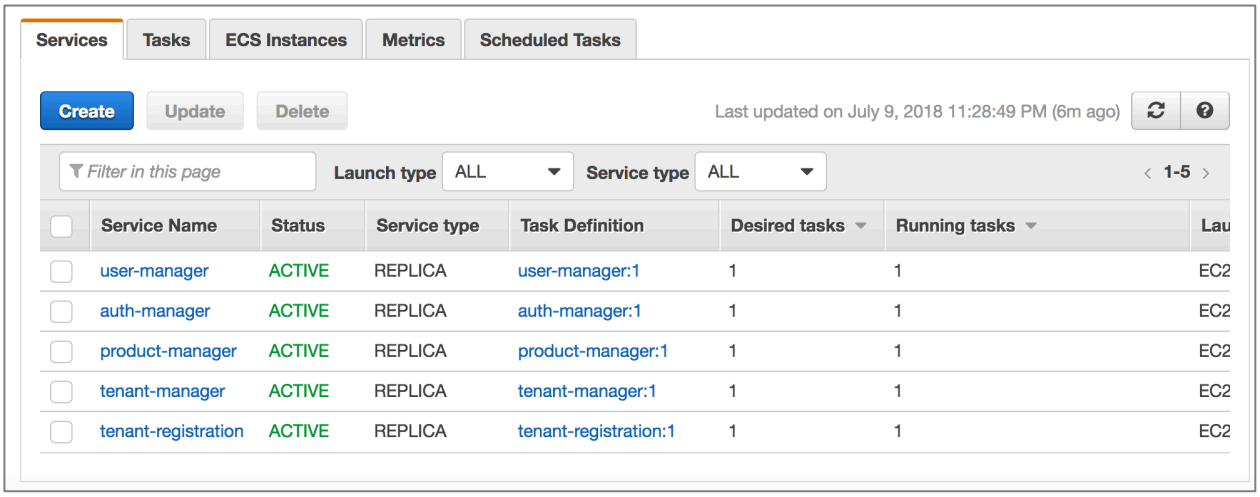
Step 5 – Our next step is to deploy version 3 of the product manager, within our Cloud9 IDE. Navigate to **Lab2/Part3/scripts** directory, right-click **product-manager-v3.sh**, and click Run to execute the shell script.



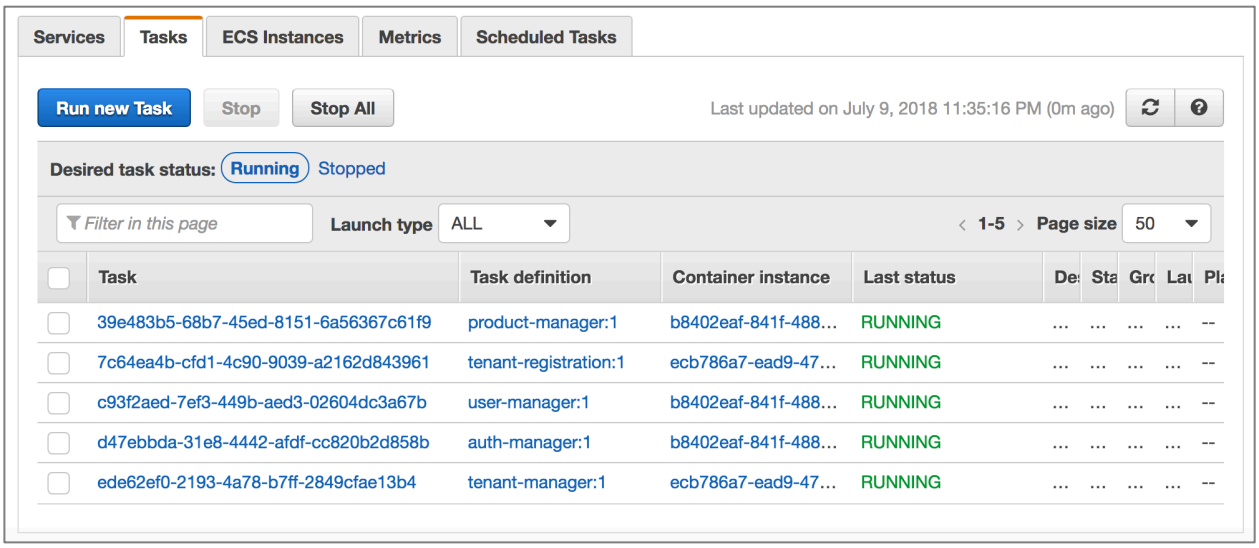
Step 6 – Wait for the **product-manager-v3.sh** shell script to execute successfully, as confirmed by the "**Process exited with code: 0**" message, and confirm that a task is properly in the "RUNNING" state by looking at the shell script output and confirming that there is at least one item in the "taskArns" the array as evaluated through the final output of the console.



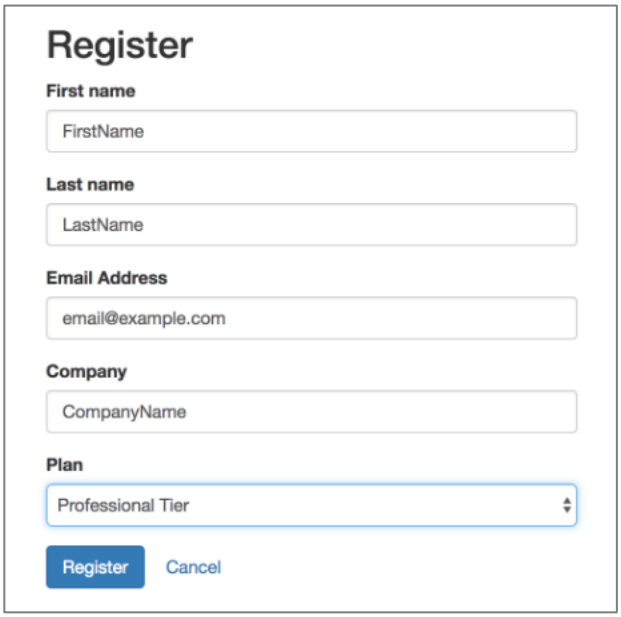
Step 7 – Navigate to the Elastic Container Service in the AWS console and note that the product management service has been created, and that the Task is in the "RUNNING" state.



Notice, that the "Task" currently matches the identifier within the "taskArn" from our shell script indicating that a new task or container has been started on Elastic Container Service with our new product-manager microservice code.



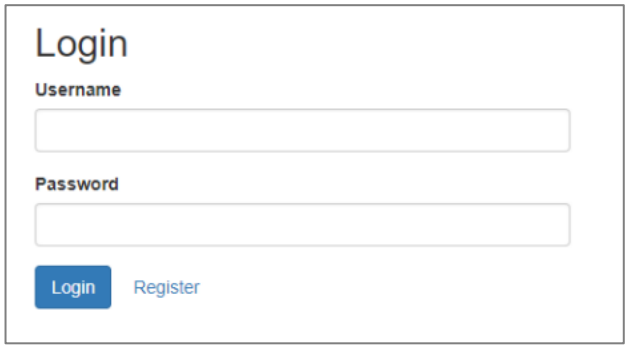
Step 8 – Now that the application is deployed, it's time to see how this new tenant and security context gets processed. In this case, we'll need to have a valid token for our service to be able to succeed. That means returning our attention to the application, which already has the ability to authenticate a user and acquire a valid token. First, we'll need to register a few new tenants through the application. Enter the URL to your application (created in Lab 1) and select the "Register" button when the login screen appears. You can get the URL for your application from the S3 bucket settings.



Step 9 – Fill in the form with data about your new tenant. Since we're creating two tenants as part of this flow, you'll need two separate email addresses. If you don't have two, you can use the same trick with the plus symbol in the username before the @ symbol as described in Lab 1. After you've filled in the form, select the "Register" button.

Step 10 – It's now time to check your email for the validation message that was sent by Cognito. You should find a message in your inbox that includes your username (your email address) along with a temporary password (generated by Cognito). The message will be similar to the following:

Step 11 – We can now login to the application using these credentials. Return to the application using the URL provided above and you will be presented with a login form (as shown below).



Step 12 – Enter the temporary credentials that were provided in your email and select the "Login" button.

Step 13 – The system will detect that this is a temporary password and indicate that you need to setup a new password for your account. To do this, application redirects you to a new form where you'll setup your new password (as shown below). Create your new password and select the "Confirm" button.

Step 14 – After you've successfully changed your password, you'll be logged into the application and landed at the home page. We won't get into the specifics of the application yet.

Step 15 – Create another tenant by repeating steps 9-14 again, supplying a different email address for your tenant.

Step 16 – As we described in Step 8, in our new version of our multi-tenant product manager microservice code, we have required an Authorization token in the header of our HTTP request. Once we logged into our website in Step 14, our token has been obtained by completing a POST method to the resource /auth of the authorization microservice we provisioned in Lab 1, which is responsible for completing an authentication to our Cognito User Pool. Once the token is returned in the content-body of our HTTP request, its included in the subsequent requests of our web application client to our microservices

Step 17 – With the web application client open in your web browser, open up the "Network" tab of your browser of choice.

Google Chrome – Click the Three Vertical Dots > "More Tools" > "Developer Tools" > "Network"

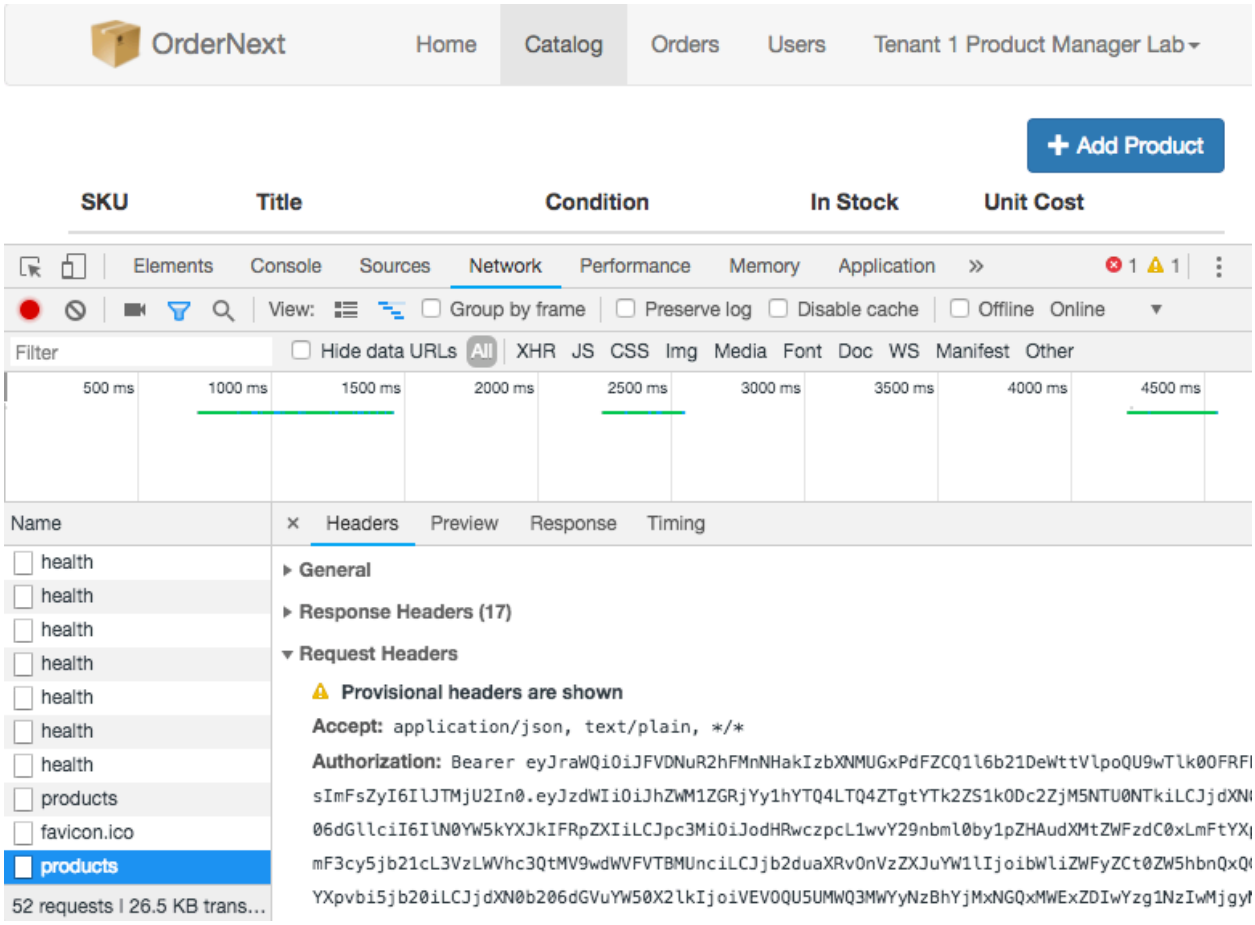
Mozilla Firefox – Click "Tools" > "Web Developer" > "Network"

Apple Safari – Safari menu > Preferences > Advanced > Show Develop menu in menu bar > Deveop > Show Web Inspector > Network tab

Step 18 – Now that our tenants have been created through the onboarding flow and we have the Network tab of our Developer Tools open, let's actually create some

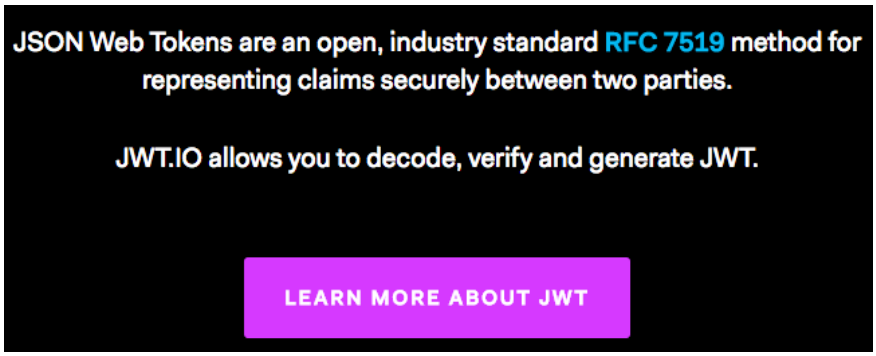
products via the application. Assuming you're still logged in as the last tenant added, you can now navigate to the "Catalog" menu item at the top of the page.

Step 19 – Look within the “Network” tab within the Browser or Web Developer Tool of choice, and click on the request navigating to the endpoint “**/products**”, and view the “Request Headers”. Find the header, labeled “Authorization” and its corresponding value.



Notice, that the “Authorization” header consists of the term “Bearer” followed by an encoded string. This is the authorization token, better known as a **JSON Web Token (JWT)** that our product manager micro-service leverages to integrate multi-tenant identity.

Step 20 – Copy the encoded token into a text editor, and open a new tab to the website https://jwt.io/. This website will allow us to decode our token to investigate the corresponding metadata within our JWT.

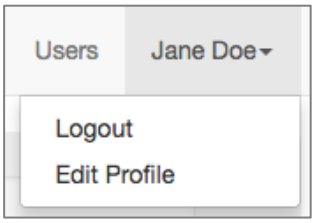


Step 21 – Scroll down the page and paste the encoded token into the "Encoded" text box in the middle of the website. This paste should have triggered a decoding of the token. Notice in the "Decoded" section of the website, the "PAYLOAD" section contains decoded key value pairs including the "email" address of the user, as well as the custom "Claims" such as custom:tenant_id which we configured as immutable within our Cognito User Pool in Lab 1.


```
{
  "sub": "aec5ddcc-aa48-48e8-a96e-d876f3955459",
  "custom:tier": "Standard Tier",
  "iss": "https://cognito-idp.us-east-1.amazonaws.com/us-east-1_pueEU0LRw",
  "cognito:username": "test@test.com",
  "custom:tenant_id": "TENANT1d71f270ab314d11a1d20c8572028262",
  "given_name": "Tenant 1",
  "aud": "29ehvlc2ae4ue58qtnnmr8ht9k",
  "event_id": "34358f2e-8c69-11e8-8539-c56504bf73e2",
  "token_use": "id",
  "auth_time": 1532124411,
  "exp": 1532128011,
  "custom:role": "TenantAdmin",
  "iat": 1532124411,
  "family_name": "SaaS Bootcamp Lab 2",
  "email": "test@test.com"
}
```


Step 22 – With the "Catalog" page open, Select the "Add Product" button to add a new product. Fill in the form with the product details and select the "Save" button. Repeat this process a few times to get a few products into the catalog for this tenant.

Step 23 – Select the drop down at the top right of the page with your tenant name and select "Logout" from the dropdown list. This will return you to the login page for the application.



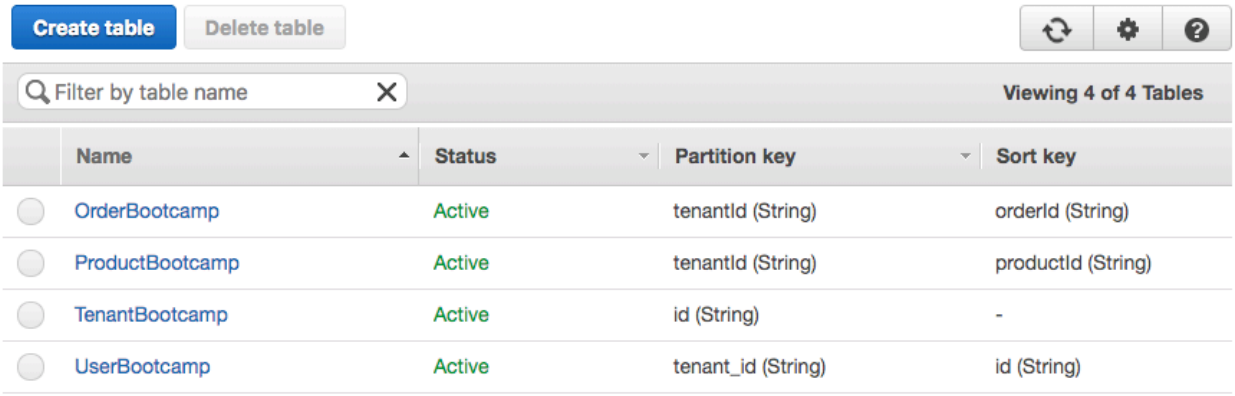
Step 24 – Enter the credentials of the other tenant that you created in step 9 above and select the "Login" button. You're now logged in as a different tenant.

Step 25 – Now navigate to the "Catalog" view again. You should note that the list of products is empty at this point. The products that you previously created were associated with another tenant so they are intentionally not show here. The illustrates that our partitioning is working.

Step 26 – Select the "Add Product" button in the products view to create a product for this tenant. Enter data for the product and select the "Save" button. Repeat this process a few times to create a few more products for this tenant.

Step 27 – After completing this onboarding process and adding these products for two separate tenants, we can now go see how this data landed in DynamoDB

tables that support this experience. Navigate to the "DynamoDB" service in the AWS console and select "Tables" from the list of options at the top left of the page. A view similar to the following will be shown.



Step 28 – Now select the "TenantBootcamp" table from the list. On the right of the page, you can now select the "Items" tab to see the list of items in your DynamoDB table (as shown here).

Step 29 – In this items list, you should be able to find the two tenants that you created through the application (separate from the tenants that were created via the command line). This verifies that our tenants were created. Note the two GUIDs for these tenants.

Step 30 – Let's now look at the product table to see the products that we created through the application. Select the "ProductBootcamp" table from the list of tables. The right-hand side of your page should still be showing the "Items" tab. Review this list of items in your product table and notice that the "tenantId" partition key is populated with the GUIDs for the two tenants that were created during registration. This illustrates that the tenant context was successfully acquired from our tokens and applied when persisting these products.

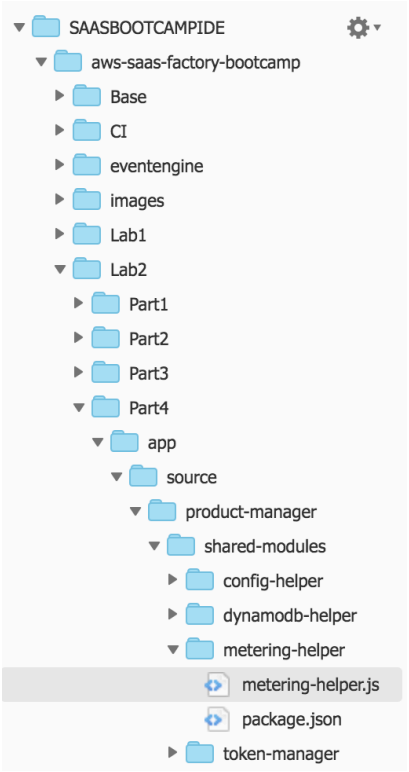
Recap: You've now elevated the mechanism of acquiring tenant context in our microservices by extracting our custom "claims" from the JWT security token passed in the Authorization HTTP header. We reduced developer complexity in applying the tenant context by creating a custom TokenManager helper class that takes advantage of the Express framework's "middleware" concept to intercept all incoming requests prior to executing a REST resource's method.

Part 4 – Applying Context to Logging, Metering, and Analytics

As part of introducing multi-tenancy into our microservice, we must also think about how tenant context will influence our approach to logging and metering of data. The idea here is that service authors should have straightforward mechanisms that allow them to emit logging and metering data complete with tenant context. This is fundamental to enabling your system's ability to diagnose and analyzing activity in the context of specific tenants.

Fortunately, solving for this is relatively simple. The main goal here is to simply introduce frameworks/libraries/modules that can be responsible for injecting tenant context into logging and metering messages. In our sample architecture we'll examine how you can push tenant aware metrics and logging data to **Amazon CloudWatch Events**, **Amazon Kinesis** data streams, and **Amazon Simple Queue Service (SQS)** for further processing.

Step 1 – We have a new version of our Product Manager microservice that has introduced important metering changes. Let's start by looking that a new **MeteringHelper** helper module that was introduced into the environment to support a centralized notion of logging and metering that enables the injection of tenant context. In Cloud9, open **/Lab2/Part4/app/source/product-manager/shared-modules/metering-helper/metering-helper.js**.



Step 2 – For **SaaS**, it is important to profile the activity and consumption of our system's resources and this is done with metering. This metering can be used

purely for analytics across the organization or parts of it can be used to inform your billing process. They key here is to capture these metrics and then figure out how they best serve the technical and business needs of the organization.

For the purposes of this bootcamp, we'll publish custom events to **CloudWatch Events** and also push those same data to a **Kinesis** data stream and an **SQS** queue. Let's look at the **putEvents()** function in our **MeteringHelper** that will accept and publish these custom metrics to CloudWatch.

```
module.exports.putEvents = function (putRecordParams, credentials) {
        var promise = new Promise(function (resolve, reject) {
            var cloudwatchevents = new AWS.CloudWatchEvents(
                {
                    apiVersion: '2015-10-07',
                    sessionToken: credentials.claim.SessionToken,
                    accessKeyId: credentials.claim.AccessKeyId,
                    secretAccessKey: credentials.claim.SecretKey,
                    region: configuration.aws_region
                });
            var Payload = JSON.stringify(putRecordParams.Payload);
            var DetailType = "MeteringRecord";
            var Source = 'ProductManager';
            var params = {
            Entries: [/* required */
                {
                    Detail: Payload,
                    DetailType: DetailType,
                    Source: Source,
                    Time: new Date()
                }
                /* more items */
            ]
            };
            cloudwatchevents.putEvents(params, function (err, data) {
                if (err) {
                    console.log('Error: Failed putEvent');
                    console.log(err);
                    reject(err);
                }
                else {
                    console.log('Success: putEvent');
                    resolve(data);
                }
            });
        });
        return promise;
};
```

Step 3 – Now let's take a look at the entry point of our MeteringHelper class – the **Record** object. By decoupling our data representation from the mechanics of processing it, we can send tenant aware data to any number of facilities for metering, analytics or logging. We have shown 3 such examples in the code connecting to CloudWatch Events, Kinesis data streams and SQS queues.

```
module.exports.Record = function (serviceParam, requestParam, credentials, error) {
    var RequestID = this.createRequestID();
    var SessionID = this.createSessionID();
    var TenantID = tokenManager.getTenantId(requestParam);
    var Service = this.setService(serviceParam);
    var Request = this.MapRequest(requestParam);
    var Context = this.MapContext(requestParam);
    var Error = this.HandleError(error);

    if(!RequestID && !SessionID && !TenantID && !Service && !Request && !Context){
    }
    else{
        var Record = this.createRecord(RequestID, TenantID, SessionID, Request, Context, Service, Error);
        if(Record === undefined || !Record){
            console.log('Create Record Error');
        }
        else{
            var params = {
                Payload: Record,
                Source: Service
            };
            this.putEvents(params, credentials);
            this.putRecord(params, credentials);
            this.sendMessage(params, credentials);
        }
    }
};
```

This **MeteringHelper** code has been simplified for this bootcamp. However, you can see how the it hides away the details of tenant injection, and is responsible for transparently obtaining the tenant context from the HTTP request.

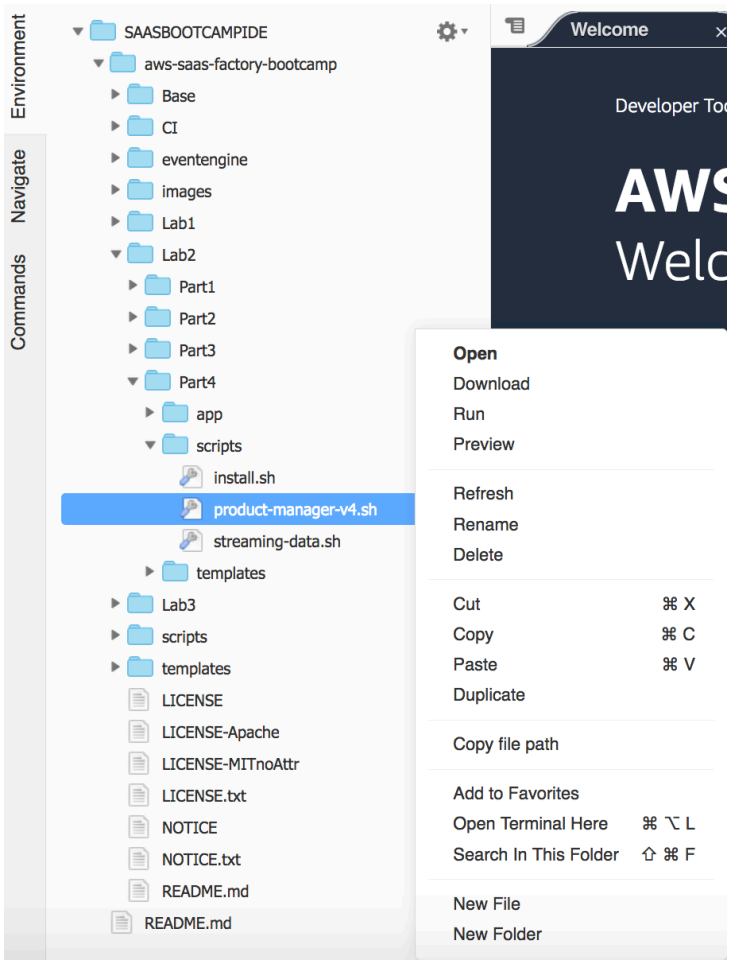
You'll notice here that the caller must send along the request and the name of the microservice as separate parameters. Then, the tenant identifier is extracted from the request programmatically by the **MeteringHelper** invoking a request to our **TokenManager** function **getTenantId()**. The new Record object created by the constructor function **Record()** will have a GUID generated, labeled **RequestId** in the record object. This GUID is leveraged as a way to identify this distinct request and tenant context to the microservice.

Step 4 – Now let's look at how this helper is utilized by the product manager service. Below is a snippet of the same method that we'd been evolving in our prior iterations. You'll notice that the we have adjusted our **GET** Method to our "**/product**" resource to incorporate our **MeteringHelper Record()** function. In addition, the parameters are now set to include either the response object of our corresponding function call to our **DynamoDBHelper**, or the error message if the request in our microservice results in a failure.

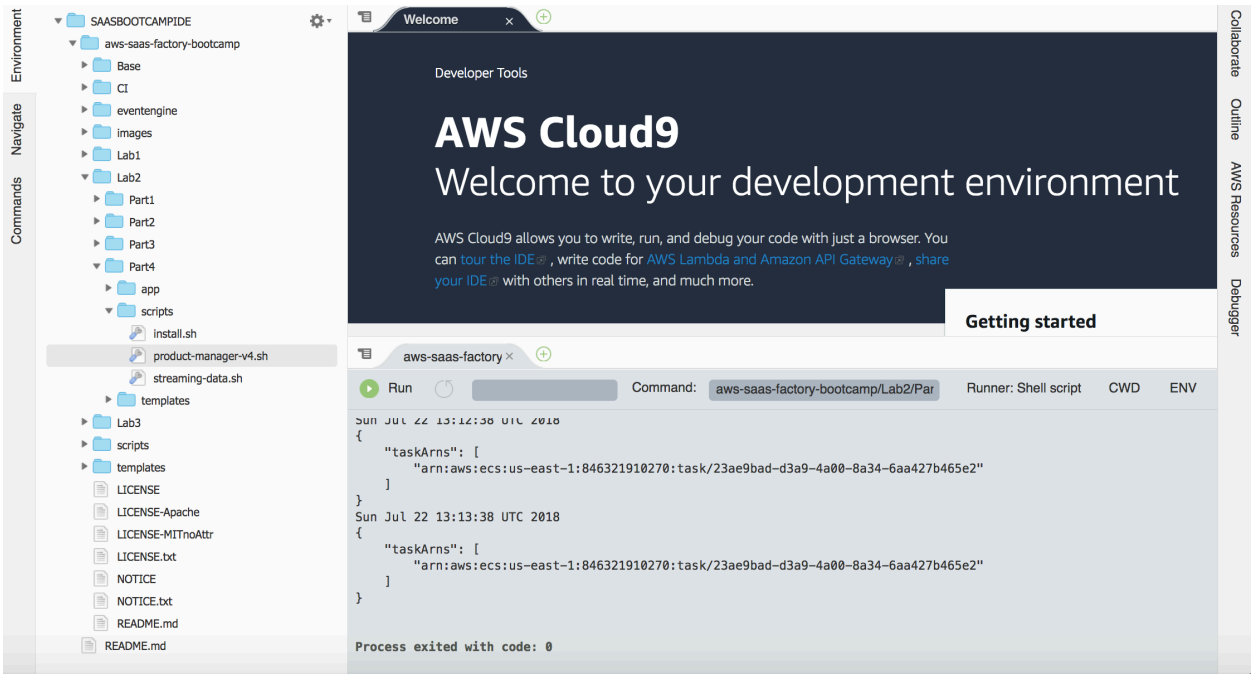
```
app.get('/product/:id', function(req, res) {
    winston.debug('Fetching product: ' + req.params.id);
     // init params structure with request params
    var params = {
        tenantId: tenantId,
        productId: req.params.id
    };
    tokenManager.getSystemCredentials(function (credentials) {
        // construct the helper object
        var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
        dynamoHelper.getItem(params, credentials, function (err, product) {
            if (err) {
                winston.error('Error getting product: ' + err.message);
                MeteringHelper.Record(configuration.name.product, req, credentials, err);
                res.status(400).send('{"Error" : "Error getting product"}');
            }
            else {
                winston.debug('Product ' + req.params.id + ' retrieved');
                MeteringHelper.Record(configuration.name.product, req, credentials);
                res.status(200).send(product);
            }
        });
    });
});
```

It is now the responsibility of our **Record()** function in our **MeteringHelper** to invoke the **put*** and **send*** functions that are responsible for ingesting our JSON object into our chosen processing streams.

Step 5 – Now that we've examined the code changes to enable collection of multi-tenant metering data let's see it in action. Our first step is to deploy the new version of our Product Manager microservice that contains our new MeteringHeper class. In Cloud9, navigate to **Lab2/Part4/scripts** directory, right-click **product-manager-v4.sh**, and click Run to execute the shell script.



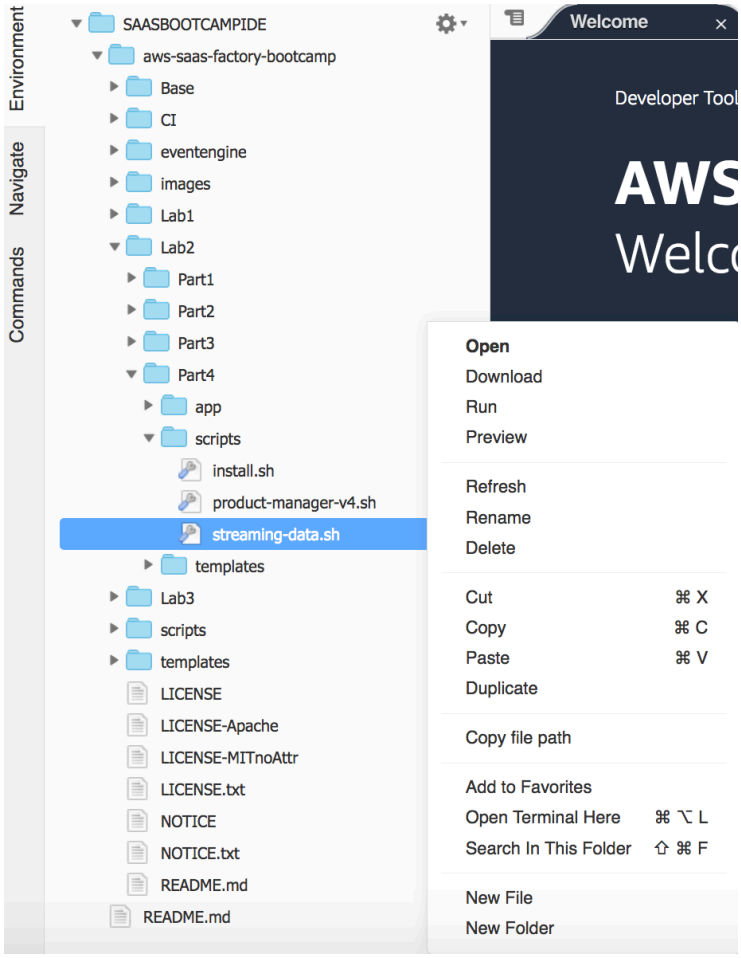
Step 6 – Wait for the **product-manager-v4.sh** shell script to execute successfully, as confirmed by the "**Process exited with code: 0**" message, and confirm that a task is properly in the "RUNNING" state by confirming that there is at least one item in the "taskArns" array as evaluated through the final output of the console.

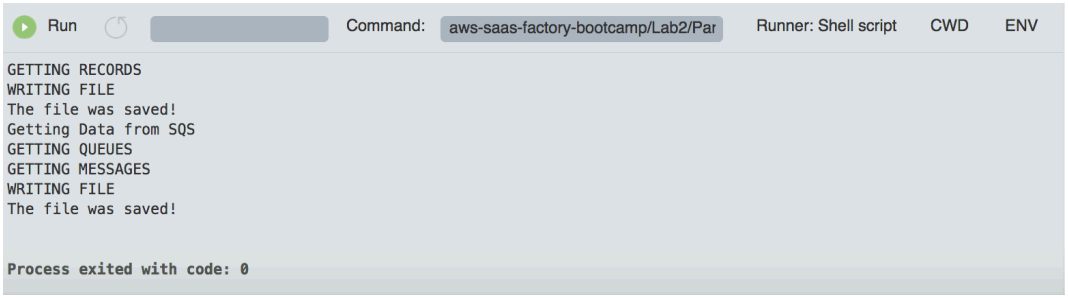


Step 7 – Now that our new Product Service has been deployed, we need to enter a couple of new products into our catalog to trigger some CloudWatch Events that will use to help us capture tenant aware metering and logging data.

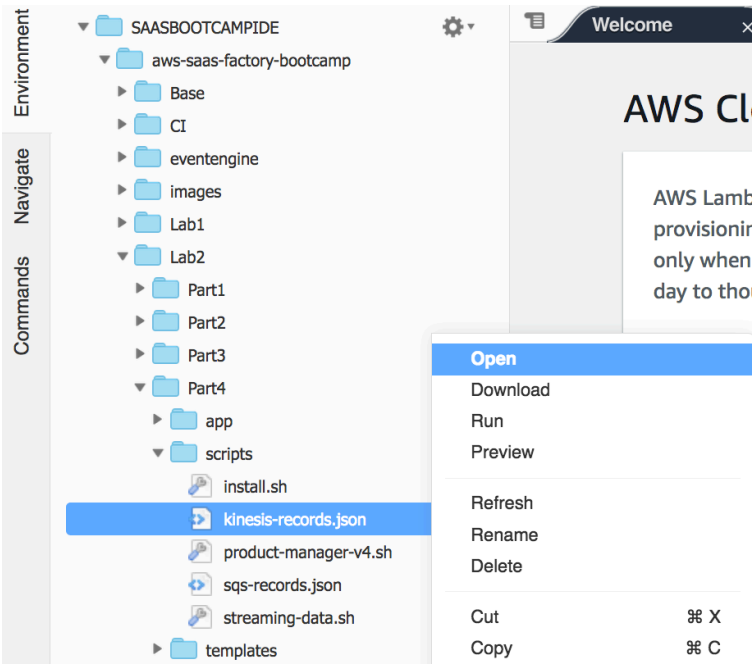
If you are still logged into the web app, log out and log back in. And, just like you did in Step 22 above, enter a couple of products into the catalog for your tenant.

Step 8 – Metering and log data was generated and pushed to AWS. Let's take a look at it. In the same scripts folder in Cloud9 (Lab2/Part4/scripts), right-click on the "streaming-data.sh" script and choose Run. This script will connect to Kinesis and SQS and download the generated metering data and save it to two local files on your Cloud9 workstation. When the script finishes successfully, you'll see "The file was saved!" and "Process exited with code: 0".



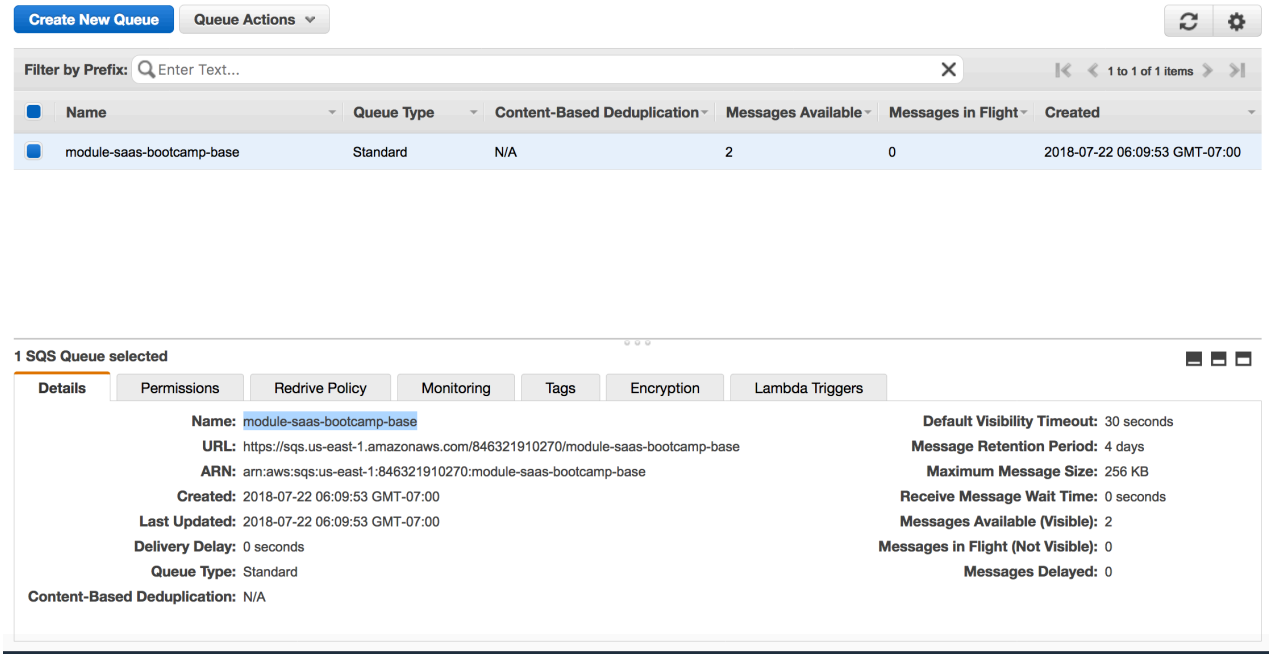


Step 9 – Notice that two new files were created in the "Lab2/Part4/scripts" directory.
a. kinesis-records.json
b. sqs-records.json

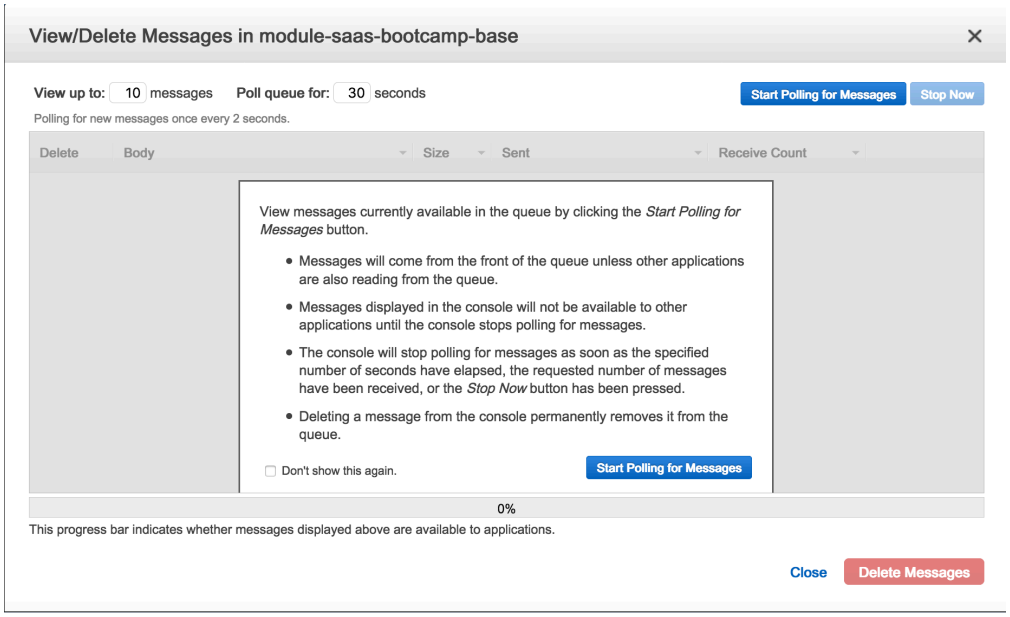


Step 10 – Open up **kinesis-records.json** and notice that the requests for POST '/product' were created and metered and sent to Kinesis Streams. These data contain tenant context. On your keyboard hold "CTRL/CMD" and press "F". In the search box, type "Tenant" and click "Enter".

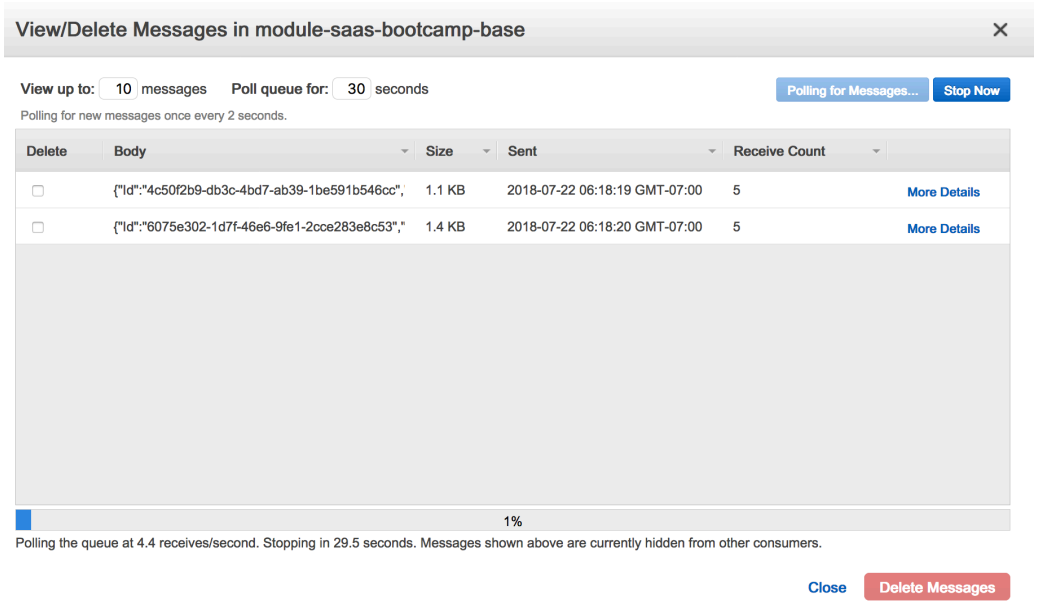
Step 11 – As an alternative, we can also view our data in SQS. Open the "**Simple Queue Service**" from the AWS console under the Application Integration heading. Select the "**module-saas-bootcamp-base**" queue.



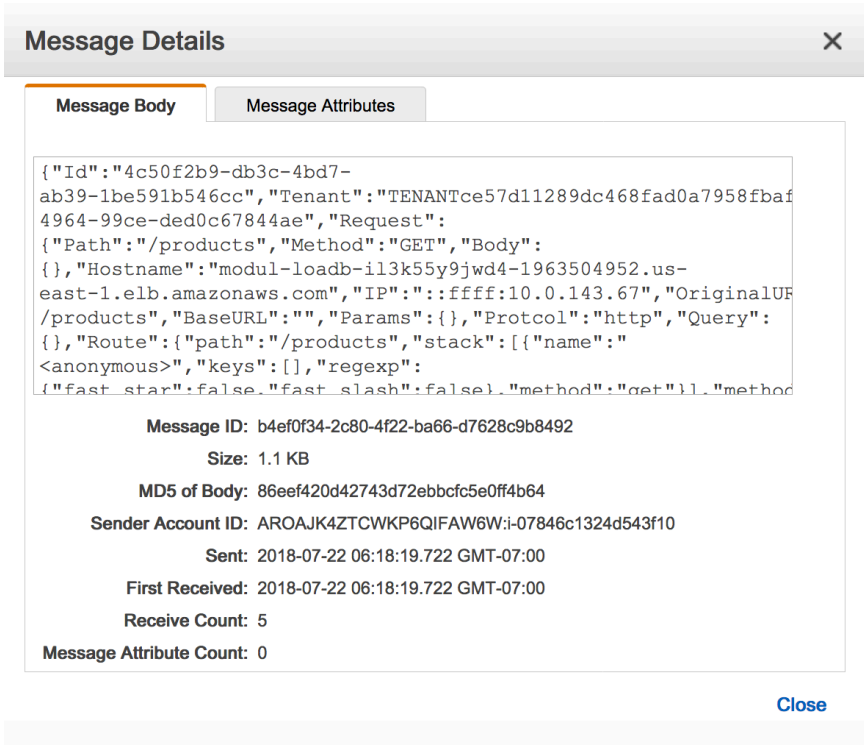
Step 12 – Click on "Queue Actions" > "View/Delete Messages" and click "Start Polling for Messages"



Step 13 – Soon after clicking the button, records will show up in the console.



Step 14 – Click on the "More Details" link in the right-most column for one of the messages in the queue and see that our MeteringHelper Record() data has also been pushed to SQS for further processing.



Step 11 – We will not be diving into how to process these metering records, however this demonstrates our analytics and metering data can be centralized in a scalable way for a custom application to process our telemetry.

Recap: the goal of this section was really to highlight the importance of injecting tenant context into your logging and metering data. It should be clear that there's no technical secret sauce needed to add tenant context. Instead, it's simply important to isolate and abstract the details of tenant injection to simplify the experience of application developers. It's also worth noting that there are many tools, frameworks, and strategies that can be employed to support and surface these metrics, and that the sample code only provides a tangible example how to add tenant context into logging and metering data.