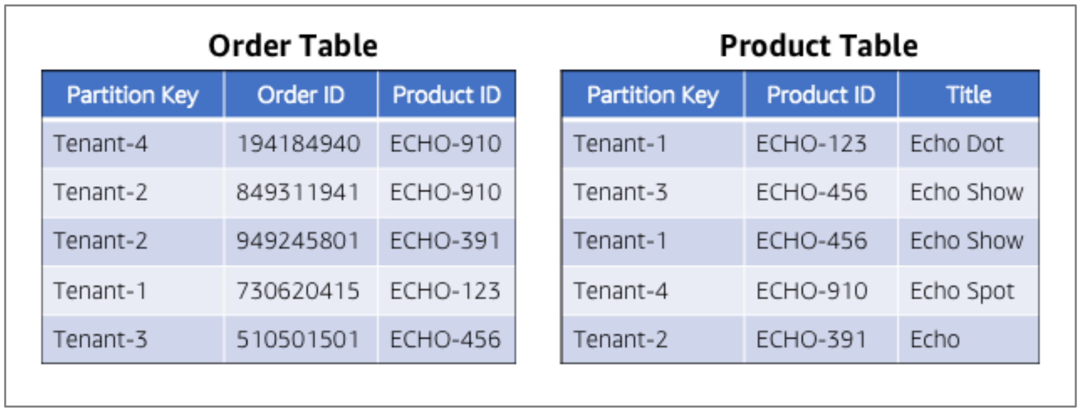
Lab 3 – Isolating Tenant Data

Overview

At this stage, we have addressed many of the core elements of SaaS architecture. What we really haven't touched on, though, is tenant isolation. As a SaaS provider, you must make every attempt to ensure that that each tenant's resources are protected from any kind of cross-tenant access. This is especially challenging when these tenants are sharing elements of their infrastructure. If, for some reason, one tenant was able to somehow access another tenant's environment, that could represent a huge setback for a SaaS business.

To address this, we must move beyond basic authentication. We must introduce policies and access controls that ensure that we are doing everything we can to isolate and protect tenant environments. Even in SaaS environments where resources are not shared, we must take extra measures to be sure that we've minimized our exposure to cross-tenant access.

For this bootcamp, we'll focus squarely on how to isolate the data that resides in our DynamoDB tables. Specifically, we want to look at how can we can successfully isolate the tenant data that resides in the product and order tables that hold application data. To achieve this, we need to consider how we've partitioned the data. Below is a diagram that highlights the partitioning scheme of the product and order tables.



Order Table

Partition Key	Order ID	Product ID
Tenant-4	194184940	ECHO-910
Tenant-2	849311941	ECHO-910
Tenant-2	949245801	ECHO-391
Tenant-1	730620415	ECHO-123
Tenant-3	510501501	ECHO-456

Product Table

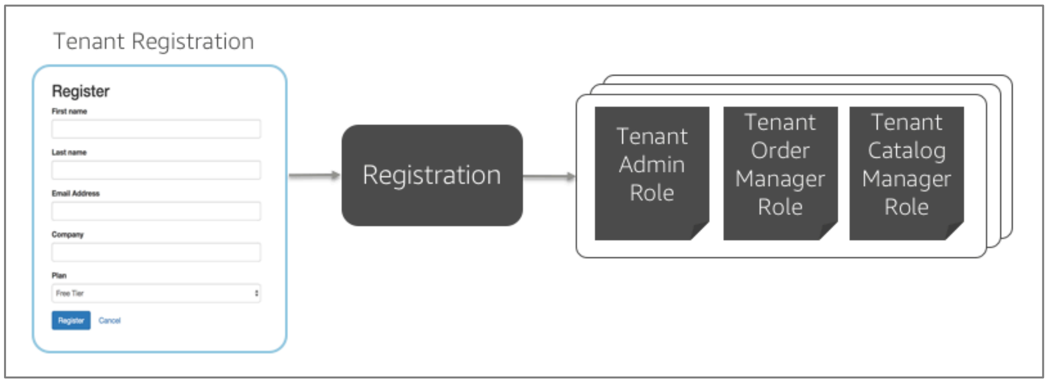
Partition Key	Product ID	Title
Tenant-1	ECHO-123	Echo Dot
Tenant-3	ECHO-456	Echo Show
Tenant-1	ECHO-456	Echo Show
Tenant-4	ECHO-910	Echo Spot
Tenant-2	ECHO-391	Echo

In these tables, you'll see that we have data from multiple tenants living side-by-side as items in these tables. So, if I get access to one of these tables, I could presumably get access to any tenant's data.

Our goal, then, is to implement a security model that can scope access to these tables down to the item level. Essentially, I want to build a view of the table that constrains access to just those items that are valid for a given tenant.

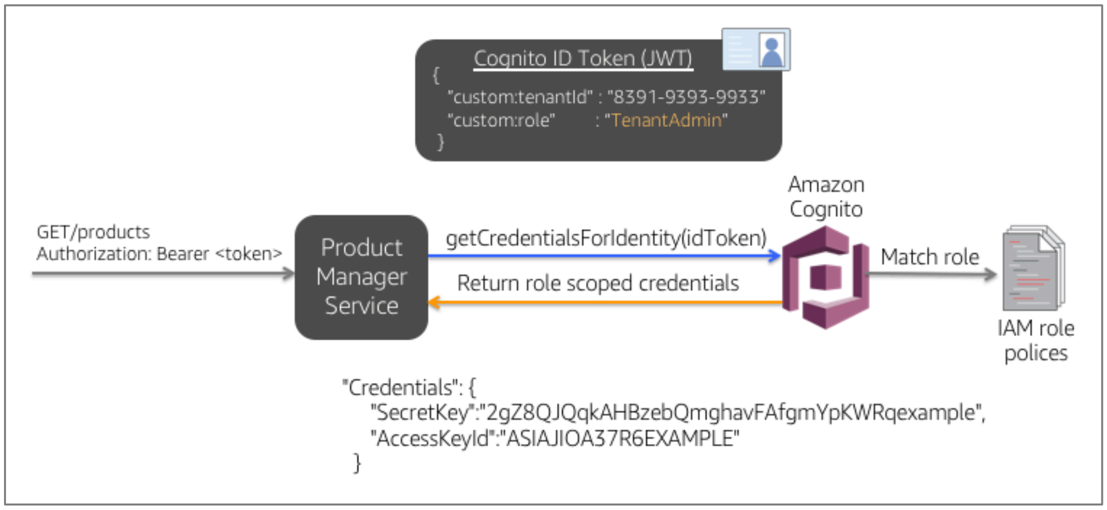
For this bootcamp, we're going to leverage a combination of Amazon Cognito, Amazon Identity and Access Management (IAM), and AWS Security Token Service (STS) to limit access to these tables. This will connect directly to the notion of SaaS identity we discussed earlier, leveraging the tokens from the experience to bind a user to a scoped set of policies.

There are two key phases to implementing this isolation model. First, when tenants are initially provisioned, we need to create a set of IAM roles for each tenant. For every role that exists in the tenant's environment, we must create policies that scope access to the system's resource for that tenant. Below you'll see a conceptual representation of this the onboarding process and how it creates these roles for each tenant.



On the left is the registration process we built in Lab 1. On the right are collections of policies that are emitted (behind the scenes) for each role. It's important to note that you are not required to have separate roles for each *user*. Instead, these roles apply to *all* users for that tenant.

The second phase of isolation comes into play when you are accessing resources from your code. The diagram below illustrates the fundamental moving parts of this process.



```
{
  "custom:tenantId" : "8391-9393-9933"
  "custom:role"      : "TenantAdmin"
}
```

```
"Credentials": {
   "SecretKey":"2gZ8QJQqkAHBzebQmghavFAfgmYpKWRqexample",
   "AccessKeyId":"ASIAJIOA37R6EXAMPLE"
 }
```

In this example, you'll see that our product manager service is invoked from the UI with a request to get a list of products. You'll note that the token (acquired during authentication) is passed along here in the Authorization header of our HTTP request. This token includes data about the user identity, role, and tenant identity. While this token is valuable for conveying user and tenant attributes, it does nothing to control a tenant's access to resources. Instead, we must use the data in this token to acquire the scoped credentials we need to access our DynamoDB tables.

The remaining bits of the diagram illustrate how these scoped credentials are acquired. Once our GET request comes into our product manager service, we'll make a getCredentialsForIdentity() call to Cognito, passing in our token. Cognito will then crack that token open, examine the tenant identifier and user role and match it to one of the policies that were created during provisioning. It will then create a temporary set of credentials (shown at the bottom) via STS and return those to our product manager service. Our service will use these temporary credentials to access DynamoDB tables with confidence that these credentials will scope access by tenant id.

What You'll Be Building

Our goal in this exercise is to walk you the configuration and creation of some of the elements that are part of this process. While the concepts are helpful above, we want to expose you to some of the specifics of how they are used in our reference solution. We'll start by introducing the policies during provisioning and how to configure Cognito to connect our policies to user roles. Lastly, we'll look at how this lands in the code of our application services. The basic steps in this process include:

- Example of Cross Tenant Access – first you'll look at how, without policies and scoping, a developer can create a situation that violates the cross-tenant boundaries of the system.
- Configure the Provisioned IAM Policies – now that you've seen an example of cross tenant access, let's start to introduce policies that can be used to prevent cross-tenant access (intended or un-intended). You'll create a policy for different role/resource combinations to get a sense of how these policies are used to scope access to DynamoDB tables. You'll then provision a new tenant and see how these policies are represented in IAM.
- Mapping User Roles to Policies – with Cognito, we can create rules that determine how a user's role will map to the policies that we've created. In this part you'll see how these policies have been configured for our tenant and user roles.
- Acquiring Tenant-Scoped Credentials – finally you'll see how to orchestrate the acquisition of credentials that are scoped by the policies outlined above. The credentials will control our access to data. You'll see how this explicitly enforces cross-tenant scoping.

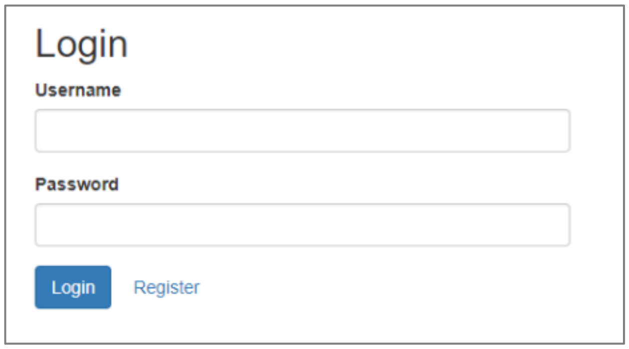
With this piece in place, you'll now have added a robust mechanism to your solution that much more tightly controls and scopes access to tenant resources. This solution highlights one of many strategies that could be applied to enforce tenant isolation.

Part 1 – Example of Cross-Tenant Access

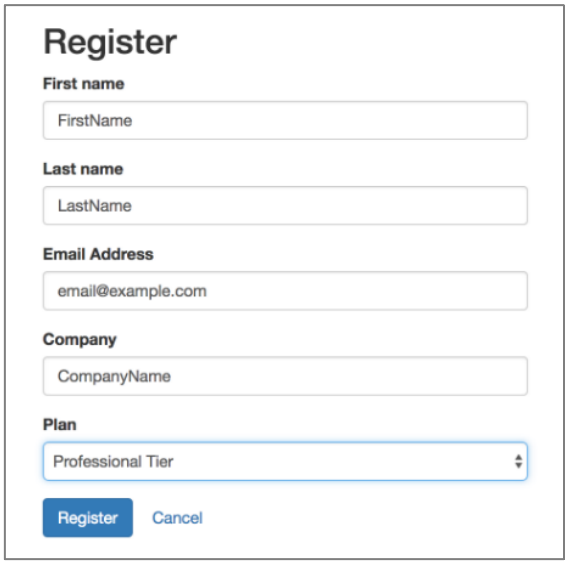
Before we introduce policies, it would help to examine a scenario where the absence of richer security policies can open the door to cross-tenant access. We will look at an admittedly contrived example where a developer could introduce code that might enable cross-tenant access.

To do this, we'll return to the product manager service and look at how manually injected tenant context could surface data in your application that should not be surfaced. This will set the stage for understanding how the introduction of policies can prevent this from happening.

Step 1 – To begin, let's return to the application and onboard a new tenant. Navigate to the application URL and you'll be placed at the login landing page. If you have a tenant that is still logged in, use the dropdown at the top right of the page log the tenant out of the system. Now, once on the login page, select the "Register" button.



Step 2 – Upon selecting the "Register" button, you'll be provided with a form where you can register your new tenant. Enter the data for your new tenant. For this tenant, the first name should be "**Tenant**" and the last name should be entered as "**One**" (the lab will refer to this tenant as "TenantOne" going forward). Also, be sure to use an email address that has *not* been previously used. Once you register the tenant you'll see a "Success" message indicating that you should check your email for further instructions.


Register

First name

FirstName

Last name

LastName

Email Address

email@example.com

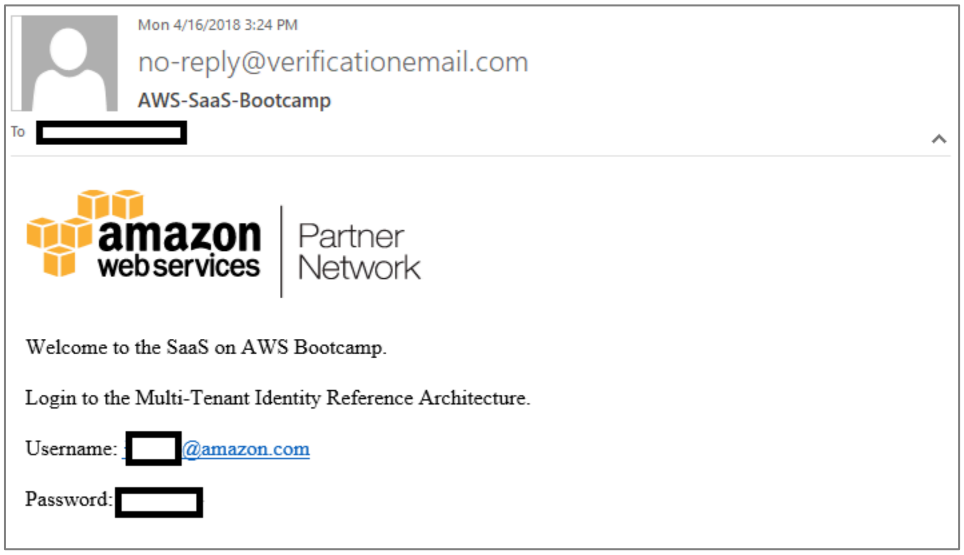
Company

CompanyName

Plan

Professional Tier

Step 3 – 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:


Mon 4/16/2018 3:24 PM

no-reply@verificationemail.com

AWS-SaaS-Bootcamp

To [redacted]

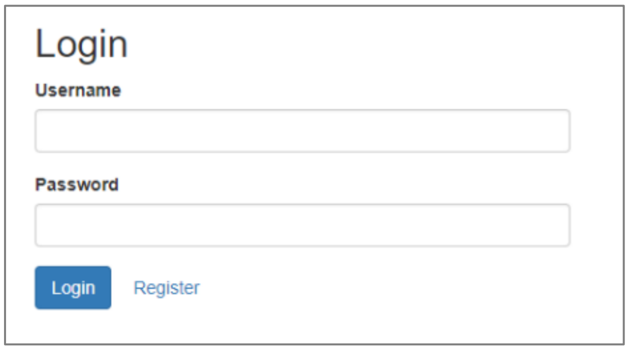
Welcome to the SaaS on AWS Bootcamp.

Login to the Multi-Tenant Identity Reference Architecture.

Username: [redacted]@amazon.com

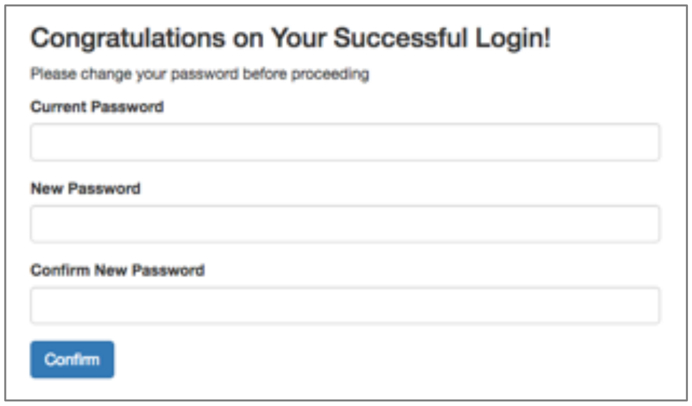
Password: [redacted]

Step 4 – 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).



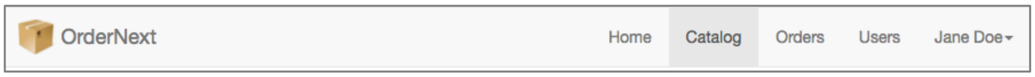
Step 5 – Enter the temporary credentials that were provided in your email and select the "Login" button.

Step 6 – Cognito 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.

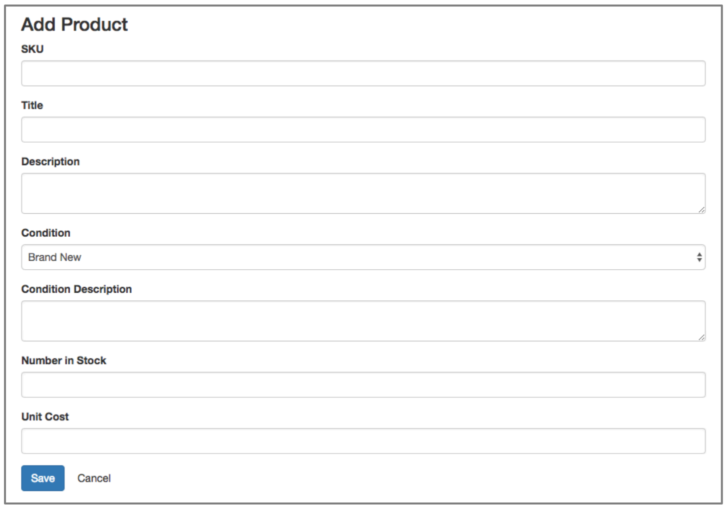

Congratulations on Your Successful Login!

Please change your password before proceeding

Step 7 – After you've successfully changed your password, you'll be logged into the application and landed at the home page. Now we can enter some data for your newly created tenant. Navigate to the "Catalog" option at the top of the page to access the product catalog.



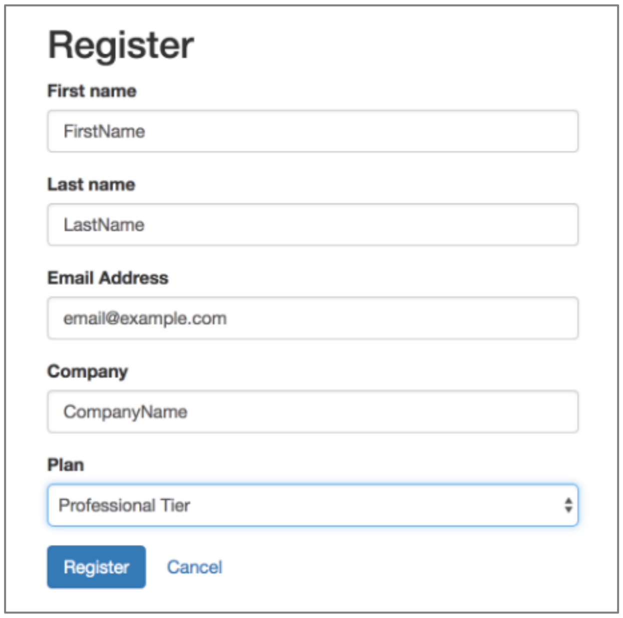
Step 8 – Now let's add some products to our catalog. Select the "Add Product" button from the top right of the page and the following form will be displayed.



Fill in the details with the product data of your choice. However, for the SKU, precede all of your SKU's with **TENANTONE**. So, SKU one might be "**TENANTONE**-ABC". The key here is that we want to have specific values that are prepended to your SKU that clearly identify the products as belonging to this specific tenant.

Step 9 – Now we need to repeat this same process for another tenant. First, we need to logout of the current tenant by selecting the dropdown from top right of the screen with your tenant name and select "Logout".

Step 10 – You'll now be placed back at the login page where you can select the "Register" button to register your new tenant. Enter the data for your new tenant. For this tenant, the first name should be "**Tenant**" and the last name should be entered as "**Two**" (the lab will refer to this tenant as "TenantTwo" going forward). Also, be sure to use an email address that has *not* been previously used. Once you register the tenant you'll see a "Success" message indicating that you should check your email for further instructions.



Register

First name

FirstName

Last name

LastName

Email Address

email@example.com

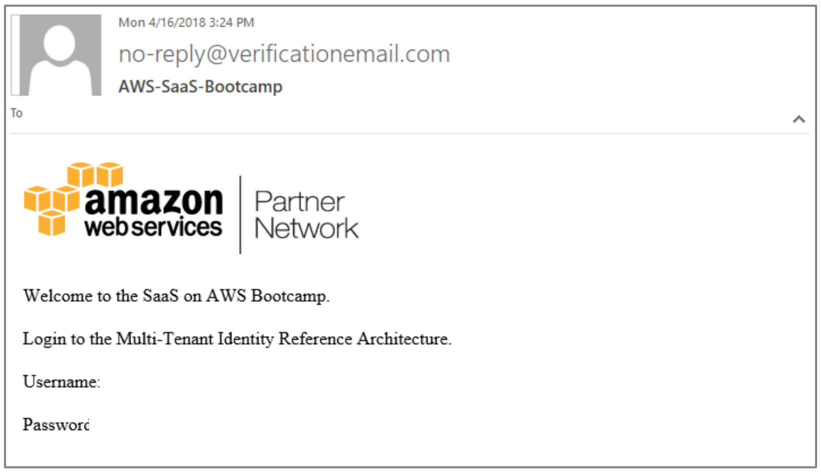
Company

CompanyName

Plan

Professional Tier

Step 11 – 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:



Mon 4/16/2018 3:24 PM

no-reply@verificationemail.com

AWS-SaaS-Bootcamp

To

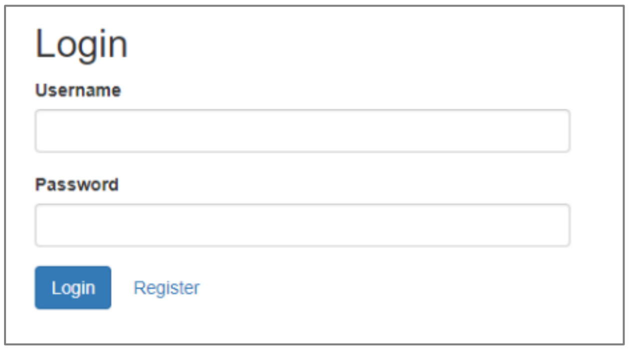
Welcome to the SaaS on AWS Bootcamp.

Login to the Multi-Tenant Identity Reference Architecture.

Username:

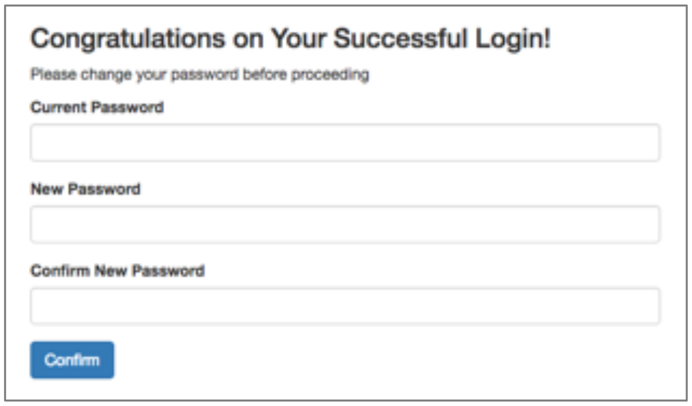
Passworc

Step 12 – 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).



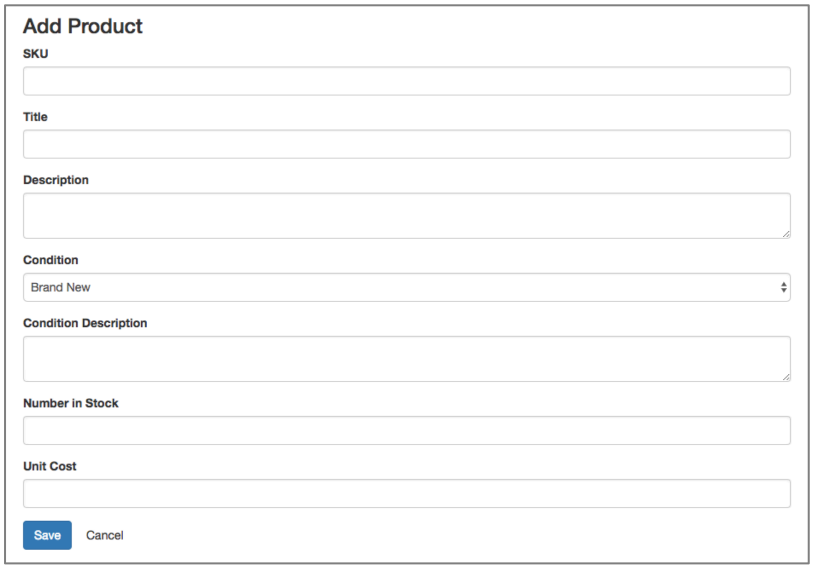
Step 13 – Enter the temporary credentials that were provided in your email and select the "Login" button.

Step 14 – Cognito 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.


Congratulations on Your Successful Login!
Please change your password before proceeding
Current Password
New Password
Confirm New Password
Confirm

Step 15 – After you've successfully changed your password, you'll be logged into the application and landed at the home page. Now we can enter some data for your newly created tenant. Navigate to the "Catalog" option at the top of the page to access the product catalog.

Step 16 – Now let's add some products to our catalog. Select the "Add Product" button from the top right of the page and the following form will be displayed.


Add Product

SKU

Title

Description

Condition

Brand New

Condition Description

Number in Stock

Unit Cost

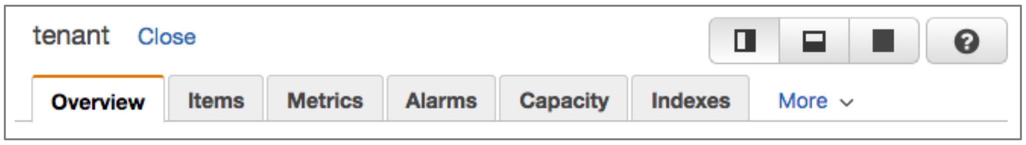
Save Cancel

Fill in the details with the product data of your choose. However, for the SKU, precede all of your SKU's with **TENANTTWO**. So, SKU one might be "**TENANTTWO**-ABC". The key here is that we want to have specific values that are prepended to your SKU that clearly identify the products as belonging to this specific tenant.

Step 17 – Let's now repeat this same process for a second tenant. First logout of the current tenant by selecting the dropdown at the top right of the page with your tenant name and selecting the "Logout" option. Now repeat steps 1-8, creating another new tenant and entering products. However, for step 8, enter products with a prepended SKU of "TENANTTWO" to identify these product as belonging to the second tenant.

Step 18 – So, now we have a new tenant with products that belong exclusively to that tenant. Let's go find the tenant identifier of this newly created tenant. Navigate to the **DynamoDB** service in the **AWS console** and select the "Tables" option located on the upper left-hand side of the page.

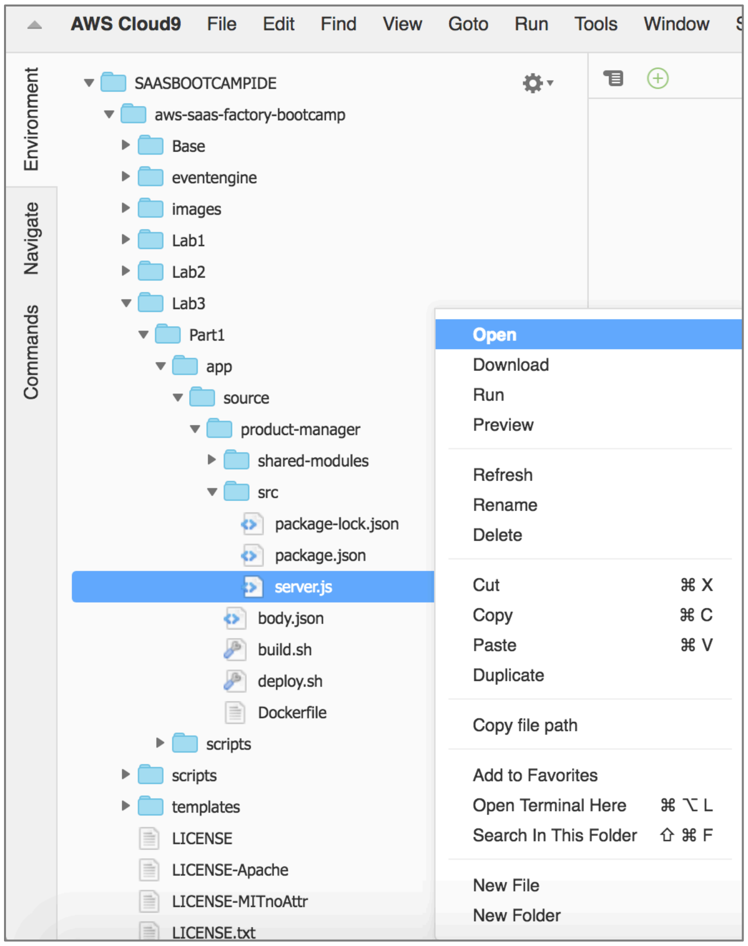
Step 19 – Select the "TenantBootcamp" table from the list of tables before selecting the "Items" tab from the top right of the page (as shown below).



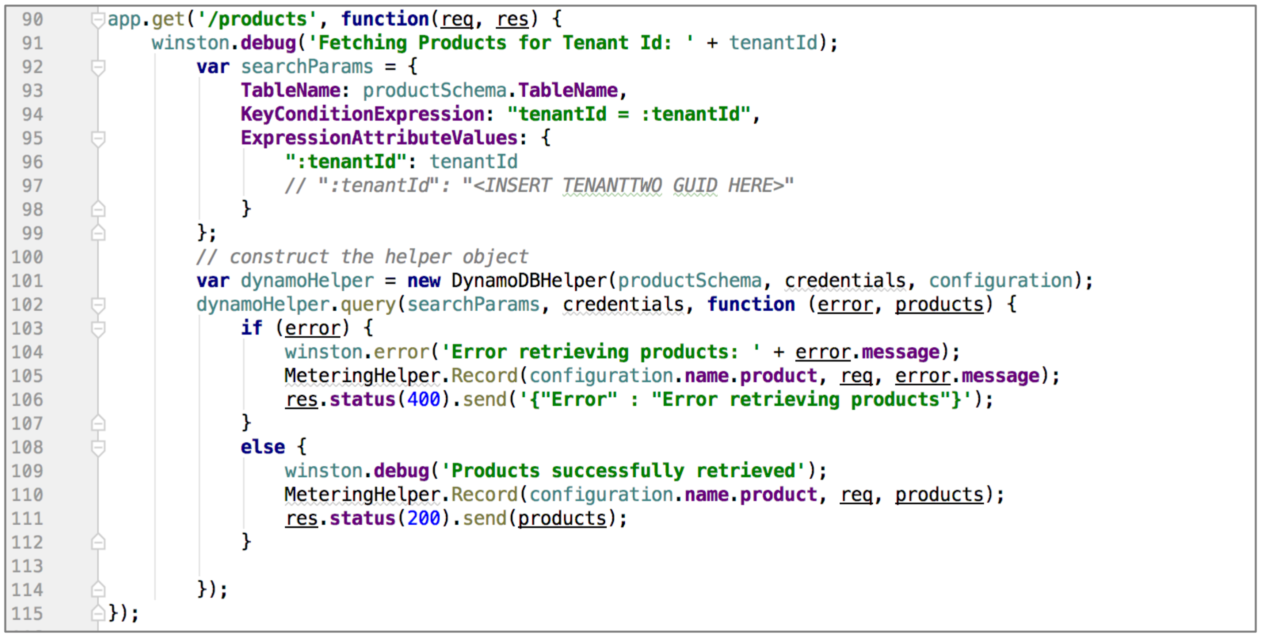
Step 20 – Locate the two tenants you created within the list by identifying the tenant with the username/email that you used above. Capture the tenant identifiers for both of these tenants.

Step 21 – Now let's go back to the code of our product manager service and make a modification.

Step 22 – Open our product manager **server.js** file in our Cloud9 IDE. In Cloud9, navigate to "**Lab3/Part1/app/source/product-manager/**", right-click **server.js** and click "Open."



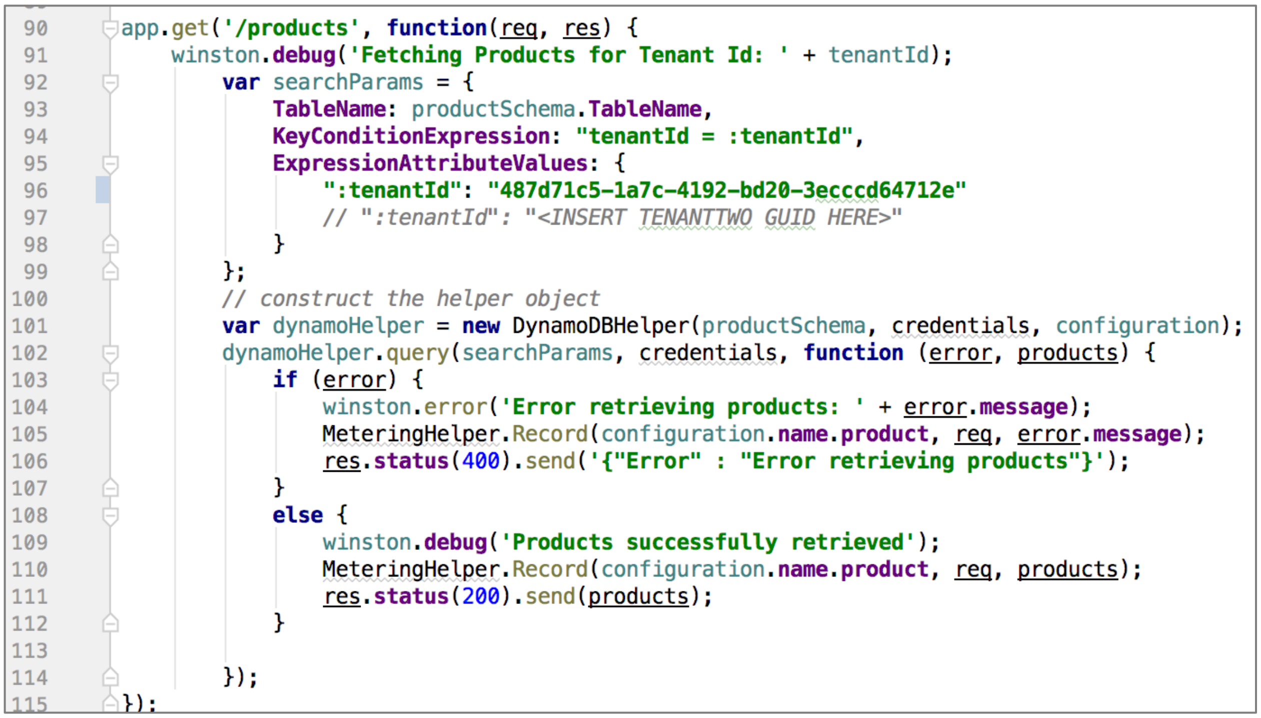
Locate the line in the function where you are fetching a list of products. The code function will appear as follows:



```
90   app.get('/products', function(req, res) {
91       winston.debug('Fetching Products for Tenant Id: ' + tenantId);
92           var searchParams = {
93               TableName: productSchema.TableName,
94               KeyConditionExpression: "tenantId = :tenantId",
95               ExpressionAttributeValues: {
96                   ":tenantId": tenantId
97                   // ":tenantId": "<INSERT TENANTTWO GUID HERE>"
98               }
99           };
100          // construct the helper object
101          var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
102          dynamoHelper.query(searchParams, credentials, function (error, products) {
103              if (error) {
104                  winston.error('Error retrieving products: ' + error.message);
105                  MeteringHelper.Record(configuration.name.product, req, error.message);
106                  res.status(400).send('{"Error" : "Error retrieving products"}');
107              }
108              else {
109                  winston.debug('Products successfully retrieved');
110                  MeteringHelper.Record(configuration.name.product, req, products);
111                  res.status(200).send(products);
112              }
113
114          });
115  });
```

This function is invoked by the application to acquire a list of products that populate the catalog page of system. You can see there on line 100 that it is references the tenantId that was extracted from the token that was passed into our application. Let's consider what might happen if were manually replace this tenantId with another value. Locate the tenantId that you recorded earlier for TenantTwo and replace the tenantId with this value. So, when you're done, it should appear similar to the following:

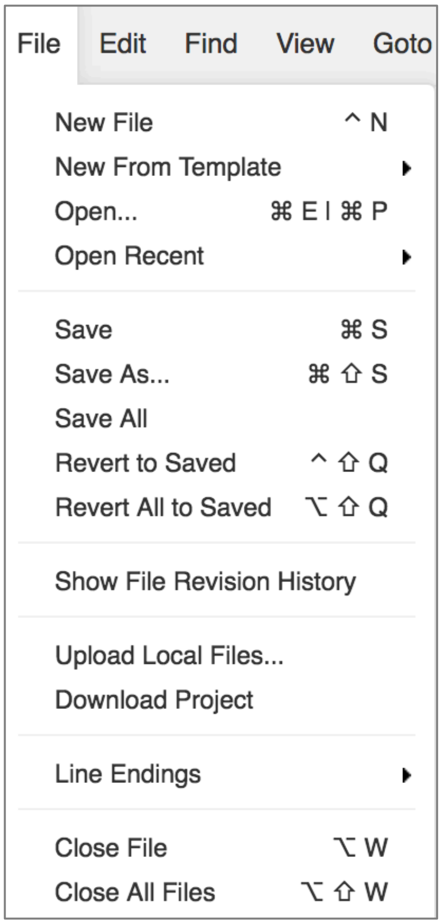
":tenantId": "<INSERT TENANTTWO GUID HERE>"



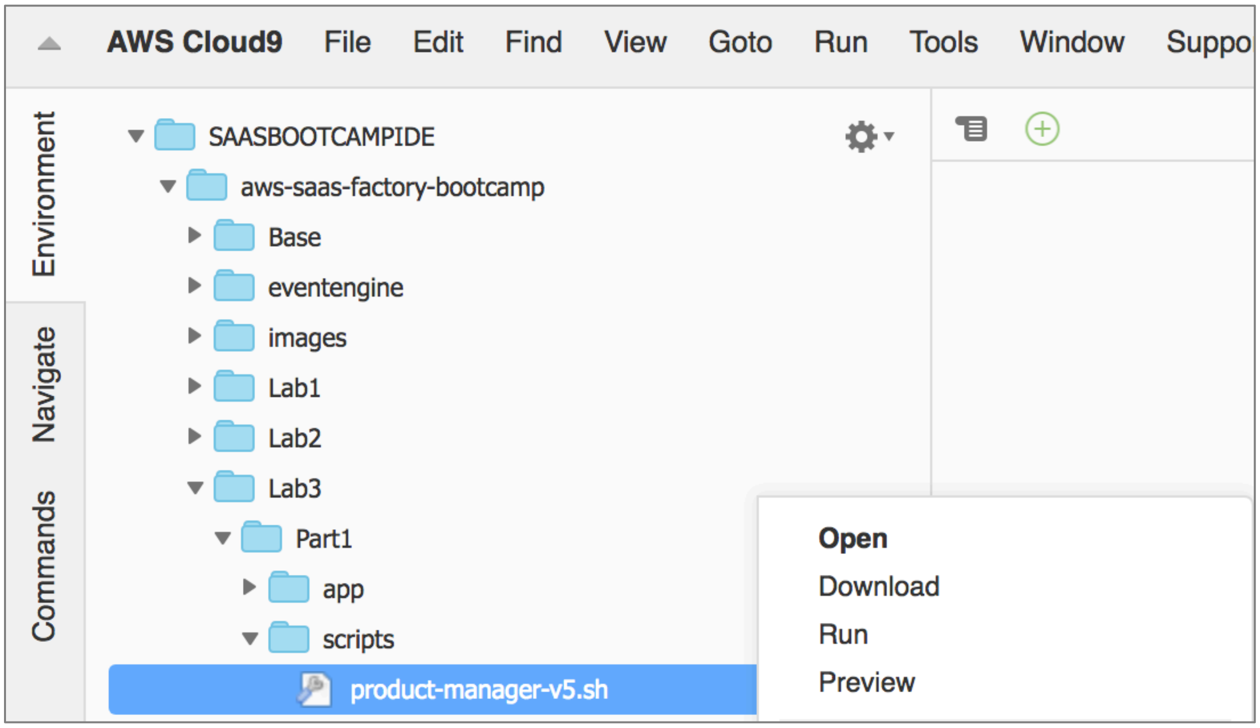
```
app.get('/products', function(req, res) {
    winston.debug('Fetching Products for Tenant Id: ' + tenantId);
        var searchParams = {
            TableName: productSchema.TableName,
            KeyConditionExpression: "tenantId = :tenantId",
            ExpressionAttributeValues: {
                ":tenantId": "487d71c5-1a7c-4192-bd20-3ecccd64712e"
                // ":tenantId": "<INSERT TENANTTWO GUID HERE>"
            }
        };
        // construct the helper object
        var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
        dynamoHelper.query(searchParams, credentials, function (error, products) {
            if (error) {
                winston.error('Error retrieving products: ' + error.message);
                MeteringHelper.Record(configuration.name.product, req, error.message);
                res.status(400).send('{"Error" : "Error retrieving products"}');
            }
            else {
                winston.debug('Products successfully retrieved');
                MeteringHelper.Record(configuration.name.product, req, products);
                res.status(200).send(products);
            }

        });
});
```

Step 23 – It is now time for us to deploy our updated product manager microservice with our cross tenant access violation in-place. First, save and replace your edited **server.js** file by clicking "File" on the toolbar, followed by "Save" as indicated in the screenshot below.



Step 24 – Our next step is to deploy version 5 of the product manager, within our Cloud9 IDE. Navigate to **Lab3/Part1/scripts** directory, right-click **product-manager-v5.sh**, and click Run to execute the shell script.



Step 25 – Wait for the **product-manager-v5.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 29 – With our new version of the service deployed, we can now see how this impacted the application. Let's log back into the system as with the credentials for **TenantOne** that you created above. Navigate to the URL for the application and enter your credentials (if the application is still logged in, log out using the dropdown at the top right of the page).

Step 29 – Select the "Catalog" menu option at the top of the page. This should display the catalog for your tenant. However, the list actually contains products that are from **TenantTwo**. We've now officially crossed the tenant boundary.

Recap: The key takeaway here is that authentication alone is not enough to protect your SaaS system. Without additional policies and authorization in place, the code of your system could un-intentionally access data for another here. Here we forced this condition more explicitly, but you can imagine how more subtle changes by a developer could have an un-intended side effect.

Part 2 – Configuring Provisioned IAM Policies

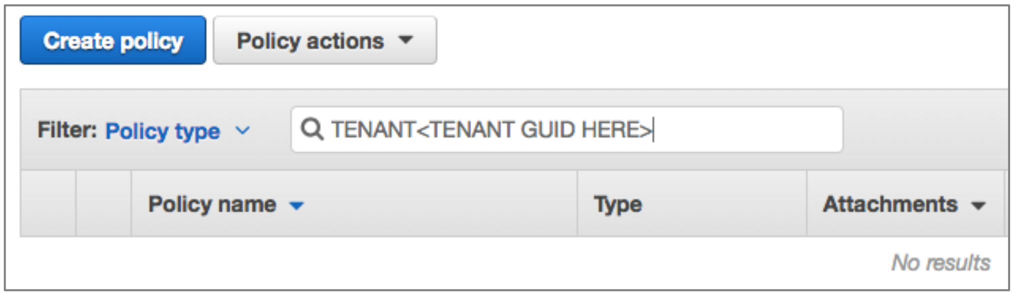
It's clear now that we need policies to better protect our system from cross-tenant access. The question is: what can we do to better isolate and protect tenant data? The first piece of the puzzle is IAM policies. With IAM policies, we can create policies that control the level of access a user has to tenant resources.

Instead of creating new policies from scratch, let's edit policies that were provisioned during the start of our process. The following steps will guide through the policy editing process:

Step 1 – To get started, we need to locate to the policies we want to edit. To get started, you'll need to navigate to the IAM service in the AWS console.

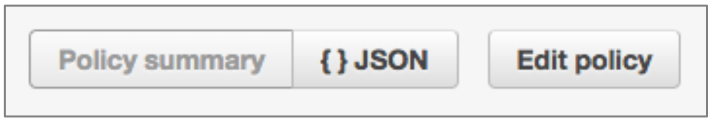
Step 2 – Next, you'll need to select "Policies" from the list of options on the upper left-hand side of the page. This will give you a list of all the polices that are available in IAM.

Step 3 – Now, we want to find the policies associated with the two tenants that we created (TenantOne and TenantTwo). Let's start with TenantOne. We need to enter the policy name in the search box near the top of the screen. In this case, it should be entered in the form of "TENANT" + the GUID of the tenant for TenantOne (which you captured earlier). The screen below provides a template for what you'll need to enter:

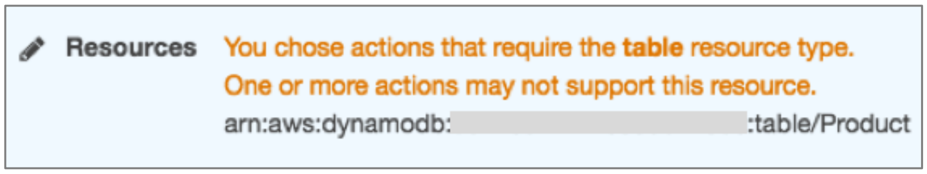


TENANT<TENANT GUID HERE>

Step 4 – The list should now be narrowed to just the 2 policies for tenant one. Select the arrow in the column preceding your policy name to drill into the policy. Then, select the "Edit policy" button that's near the center of the page.



Step 5 – The console will now display a list of DynamoDB polices (and a Cognito User Pool policy). We're interested in editing the policy for the product table. However, it's location in this list of DynamoDB tables can vary. Open each of the collapsed DynamoDB entries in this list by select the arrow at the left edge of the list. Near the bottom of each expanded set of polices, you should find a "Resources" section. Locate the set of policies that reference the "**ProductBootcamp**" table. The ARN will be similar to the following:

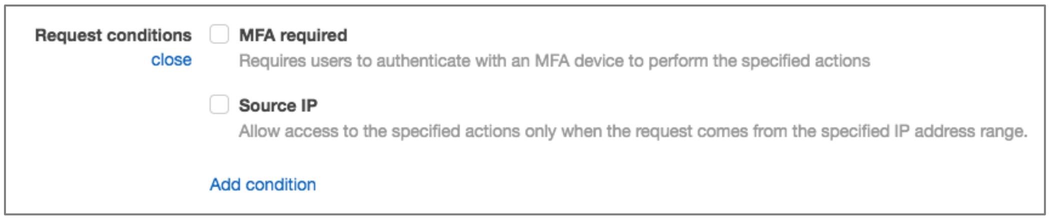


Resources You chose actions that require the **table** resource type. One or more actions may not support this resource.

arn:aws:dynamodb:[redacted]:table/Product

Step 6 – Our interest is in the "Request conditions" associated with this policy. These conditions are at the heart of our ability to control which items a user can access within a DynamoDB table. We want our policy to indicate that *only* users with partition key value that matches TenantOne's tenant identifier will be allowed to access those items in the table. Hover over the "Request conditions" value and select the text for the conditions this will put you into edit mode for the conditions.

Step 7 – When the request conditions expands, you'll see the following:



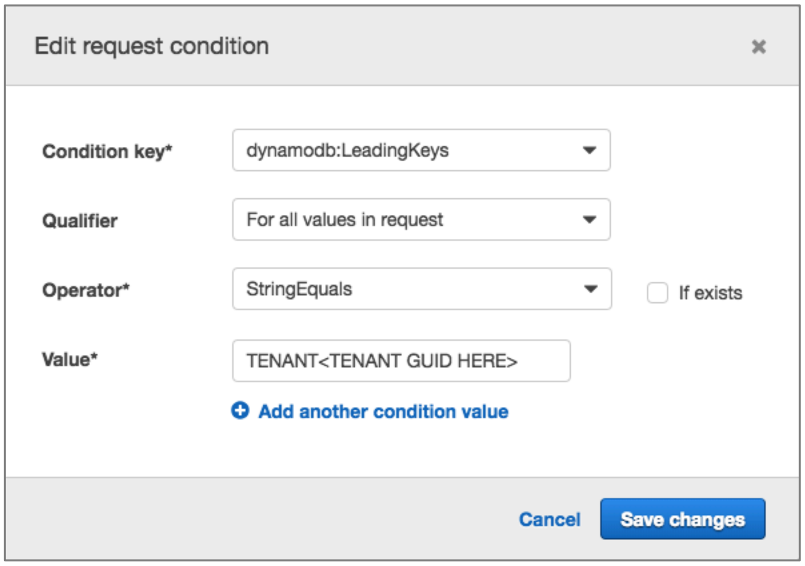
Request conditions
close

☐ **MFA required**
Requires users to authenticate with an MFA device to perform the specified actions

☐ **Source IP**
Allow access to the specified actions only when the request comes from the specified IP address range.

Add condition

Step 8 – Select the "add condition" option at the bottom of the list and that will display the following form:


Edit request condition

Condition key*	dynamodb:LeadingKeys
Qualifier	For all values in request
Operator*	StringEquals [ ] If exists
Value*	TENANT<TENANT GUID HERE>

Select "**dynamodb:LeadingKeys**" for the Condition key. Select "**For all values in request**" for the Qualifier. Selection "**StringEquals**" for the Operator. Finally, in the Value, enter "TENANT" + the GUID of TenantOne. Select the "Save Changes" button to save this change to the policy.

This process created a new request condition for our policy that now indicates that the value of our partition key in our DynamoDB table must include the tenant identifier when you user attempts to access items in the table.

Step 9 – We now want to repeat this same process for **TenantTwo**. Complete steps 2-8 again replacing all reference to **TenantOne** with **TenantTwo**. This will ensure that **TenantTwo** is also protected.

Recap: Our goal here was to begin to put in place the elements needed to support our tenant isolation goals. This first step was all about amending our tenant policies and introducing changes that would allow us to scope access to DynamoDB tables. This was achieved by adding a new condition to our product table policies. These policies, which are tenant-specific, limit a user's view of the table to only those items that contain our tenant identifier in the table's partition key.

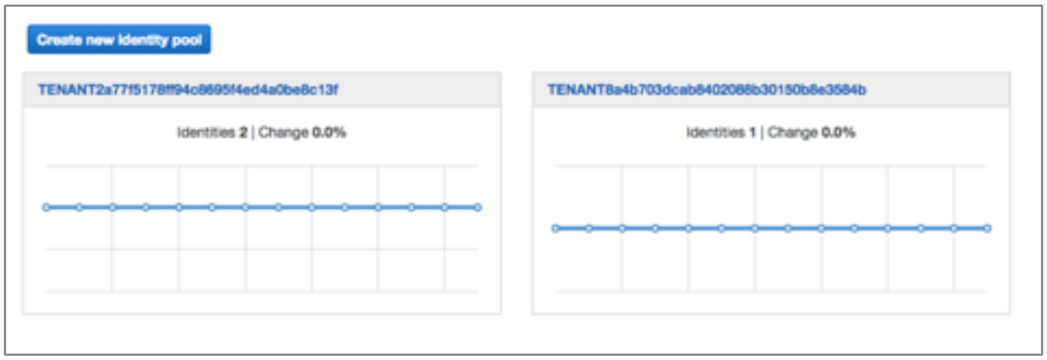
Part 3 – Mapping User Roles to Policies

Now that we have policies defined, we need some way to connect these policies with specific user roles. Ultimately, we need a way to match both the role of the user and the tenant scope to a specific set of policies. For this scenario, we're going to lean on the role matching capabilities of Cognito. Cognito will allow us to define a set of conditions that will be used to create this match and, in the process, emit a set of credentials that will be scope based on the matching policies—which is exactly what we need to implement our tenant isolation model.

In this bootcamp these policy mappings have already been created. Let's take a look at them in the Cognito console.

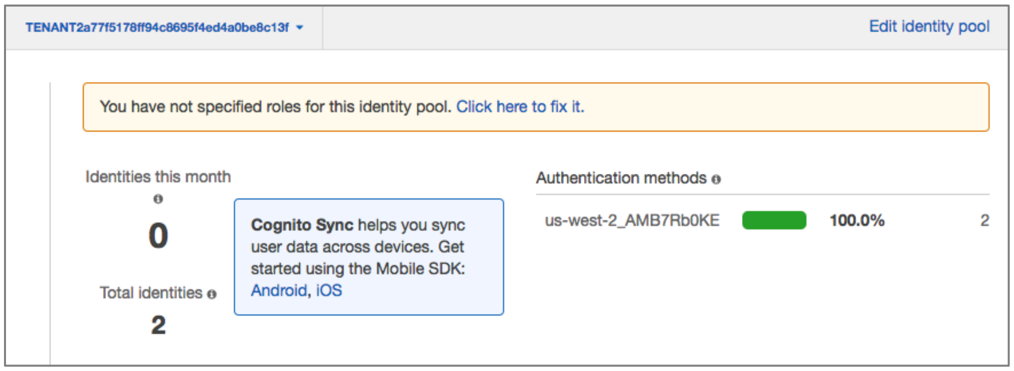
Step 1 – Navigate to the Cognito service in the AWS console. From the landing page, select the "Manage Identity Pools" button.

Step 2 – You'll now land at a page with a list of identity pools. It will Include separate pools for each of the tenants that you have onboarded. The image below provides a sample of the page:


Create new identity pool
TENANT2a77f5178ff94c8695f4ed4a0be8c13f
Identities 2 | Change 0.0%
TENANT8a4b703dcab8402068b30150b8e3584b
Identities 1 | Change 0.0%

From this page, you'll want to locate the identity pools for TenantOne and TenantTwo. They will be denoted by the name "TENANT" + the GUID of the tenant. Select the title of the identity pool that is associated with TenantOne.

Step 3 – Once you select the identity pool, you see a page that provides and overview of the identity pool activity. Now select the "Edit Identity Pool" link at the top right of the page.



You have not specified roles for this identity pool. Click here to fix it.

Identities this month 0

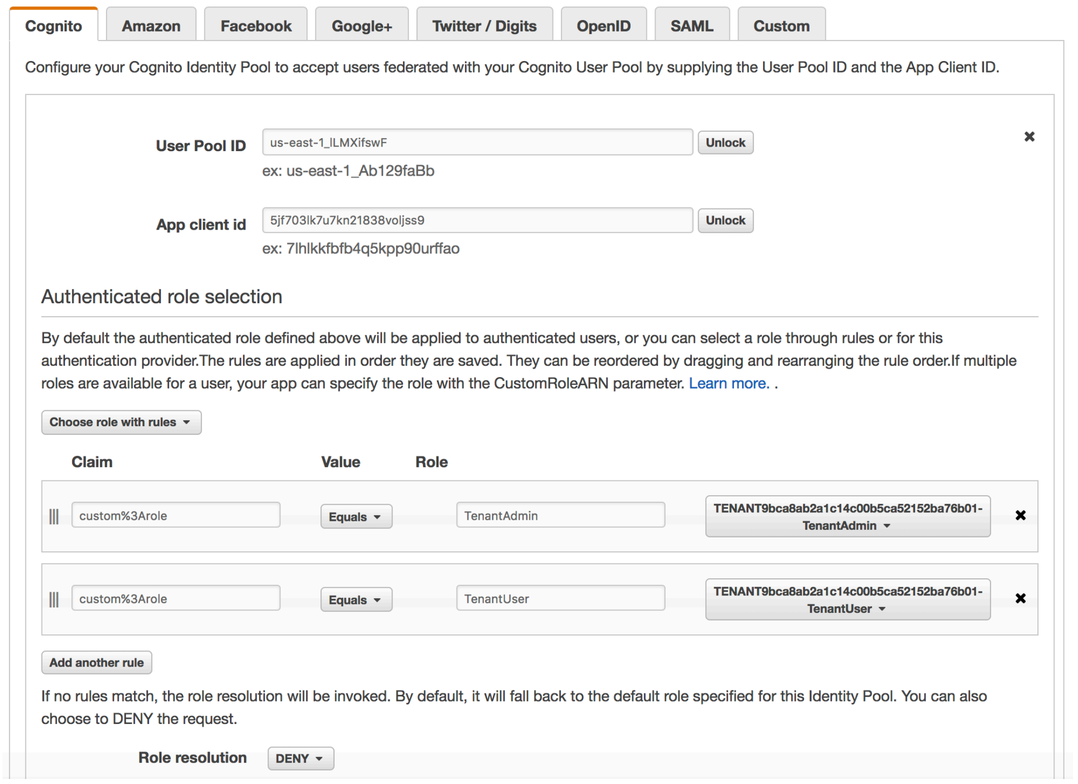
Total identities 2

Cognito Sync helps you sync user data across devices. Get started using the Mobile SDK: Android, iOS

Authentication methods			
us-west-2_AMB7Rb0KE	🟩	100.0%	2

Step 4 – If you scroll down the edit identity pool page, you'll see a heading for "**Authentication Providers**". Expand this section and you'll see a page with authorization provider configurations.

Step 5 – We can now see the role mappings in place for our two roles. There is a "TenantAdmin" role that represents the administrator and there's a "TenantUser" role that maps to individual non-admin users of your SaaS system. Naturally, these have different levels of access to the system and its resources. The claim column has a value (URL encoded) that matches the custom "role" attribute you configured in Cognito back in Lab 1. When that custom claim matches the name of the role, the IAM policy (with the DynamoDB restrictions) is enforced on the temporary security tokens returned from STS.



Configure your Cognito Identity Pool to accept users federated with your Cognito User Pool by supplying the User Pool ID and the App Client ID.

User Pool ID	us-east-1_lLMXifswF	Unlock
	ex: us-east-1_Ab129faBb	
App client id	5jf703lk7u7kn21838voljss9	Unlock
	ex: 7lhlkkfbfb4q5kpp90urffao	

Authenticated role selection

By default the authenticated role defined above will be applied to authenticated users, or you can select a role through rules or for this authentication provider.The rules are applied in order they are saved. They can be reordered by dragging and rearranging the rule order.If multiple roles are available for a user, your app can specify the role with the CustomRoleARN parameter. Learn more. .

Choose role with rules

Claim		Value	Role
custom%3Arole	Equals	TenantAdmin	TENANT9bca8ab2a1c14c00b5ca52152ba76b01-TenantAdmin
custom%3Arole	Equals	TenantUser	TENANT9bca8ab2a1c14c00b5ca52152ba76b01-TenantUser

Add another rule

If no rules match, the role resolution will be invoked. By default, it will fall back to the default role specified for this Identity Pool. You can also choose to DENY the request.

Role resolution DENY

Recap: You've now completed building out the second phase of our tenant isolation. With this exercise, we saw the role-mapping rules in our Cognito identity pool. These mappings directly associate roles for tenants (**TenantAdmin** and **TenantUser**) to the policies that we configured in first part of this lab.

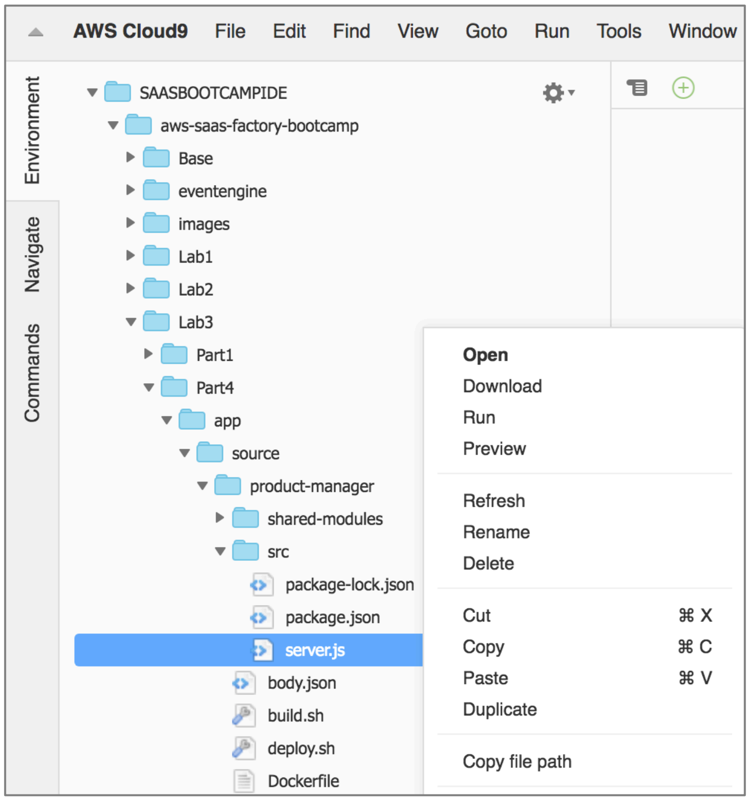
Part 4 – Acquiring Tenant-Scoped Credentials

At this point, all the elements of our isolation scheme are in place. We have authentication with Cognito, roles provisioned for each tenant that scope access to our

DynamoDB tables, and we have role-mapping conditions configured in Cognito that will connect our authenticated users with their corresponding policies. All that remains now is to introduce the code into our application services that exercises these elements and acquires credentials that will properly scope our access to the tenant resources.

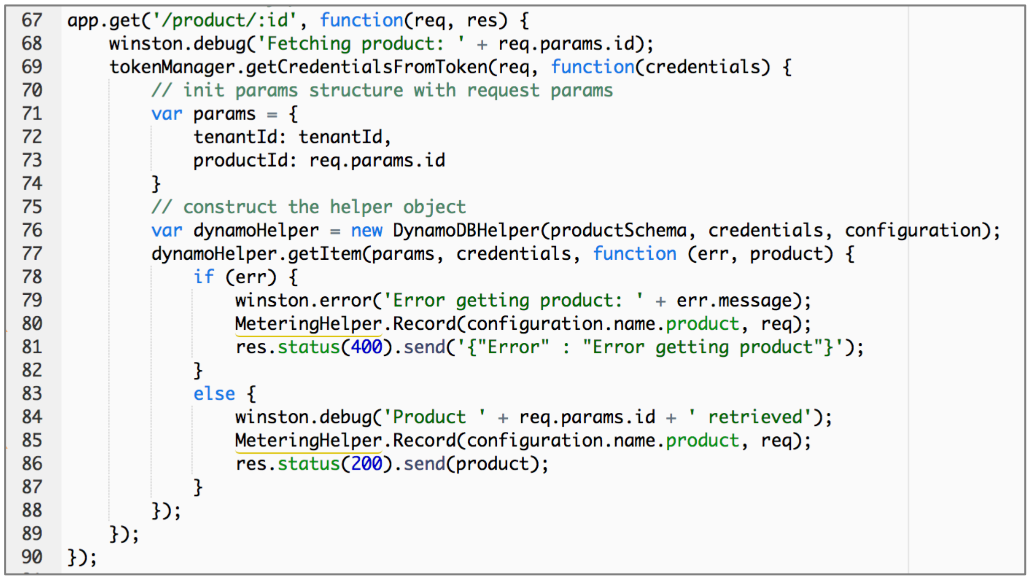
The steps that following will guide you through the process of configuring and deploying a new version of the product manager service that successfully acquires these tenant-scoped credentials.

Step 1 – Before we can get started, we need to look at how the product manager service is modified to support tenant isolation. In Cloud9, navigate to "**Lab3/Part4/app/source/product-manager/**", right-click **server.js** and click "Open."



The code shown below highlights the last key piece of the tenant isolation puzzle. On line **69**, you'll notice that we have added a call that acquires credentials form the user's token. The **getCredentialFromToken()** method accepts the request and returns the credentials that are scoped by tenant. These credentials are then used in our calls to the **DynamoDBHelper** to ensure that we cannot cross tenant boundaries.

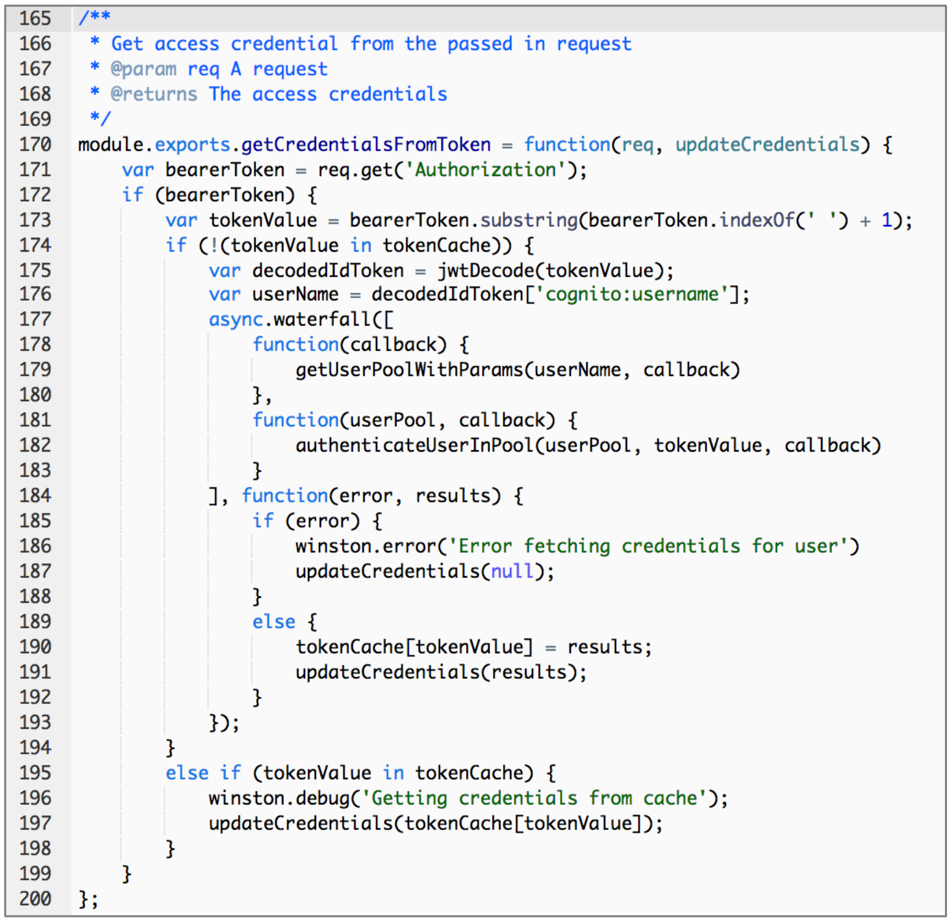
Step 2



```
app.get('/product/:id', function(req, res) {
    winston.debug('Fetching product: ' + req.params.id);
    tokenManager.getCredentialsFromToken(req, function(credentials) {
        // init params structure with request params
        var params = {
            tenantId: tenantId,
            productId: req.params.id
        }
        // 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);
                res.status(400).send('{"Error" : "Error getting product"}');
            }
            else {
                winston.debug('Product ' + req.params.id + ' retrieved');
                MeteringHelper.Record(configuration.name.product, req);
                res.status(200).send(product);
            }
        });
    });
});
```

–

The call to **getCredentialsFromToken()** described above is where all the magic happens in terms of mapping our token/identity to the appropriate policies and returning that in the form of credentials. Given the importance of this function, let's dig in and look more closely at what it is doing. Below is a snippet of code from the TokenManager that implements the **getCredentialsFromToken()** function:

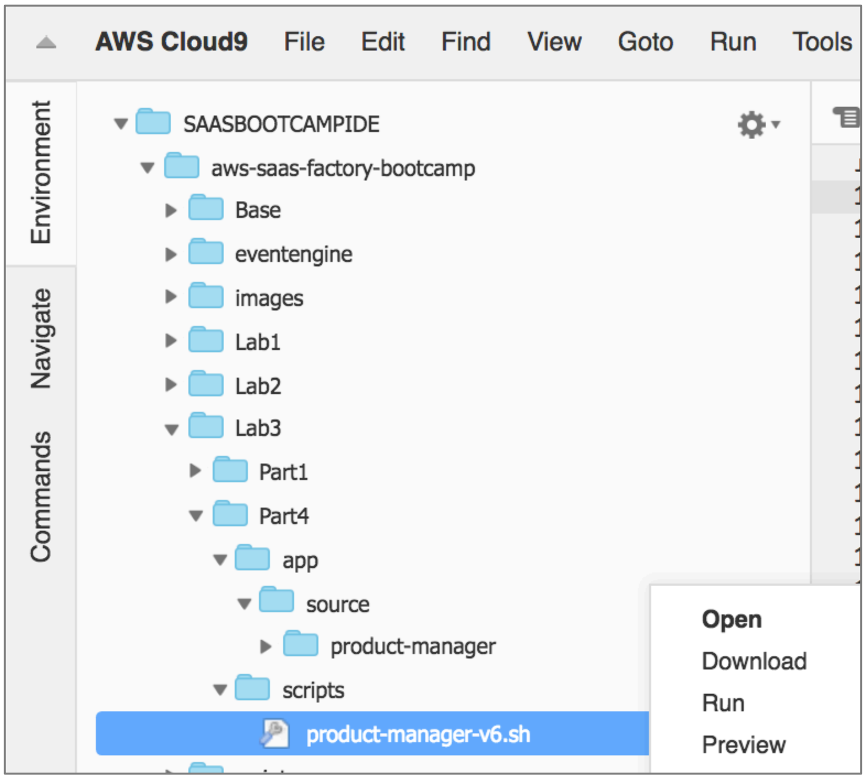


```
165 /**
166  * Get access credential from the passed in request
167  * @param req A request
168  * @returns The access credentials
169  */
170 module.exports.getCredentialsFromToken = function(req, updateCredentials) {
171     var bearerToken = req.get('Authorization');
172     if (bearerToken) {
173         var tokenValue = bearerToken.substring(bearerToken.indexOf(' ') + 1);
174         if (!(tokenValue in tokenCache)) {
175             var decodedIdToken = jwtDecode(tokenValue);
176             var userName = decodedIdToken['cognito:username'];
177             async.waterfall([
178                 function(callback) {
179                     getUserPoolWithParams(userName, callback)
180                 },
181                 function(userPool, callback) {
182                     authenticateUserInPool(userPool, tokenValue, callback)
183                 }
184             ], function(error, results) {
185                 if (error) {
186                     winston.error('Error fetching credentials for user')
187                     updateCredentials(null);
188                 }
189                 else {
190                     tokenCache[tokenValue] = results;
191                     updateCredentials(results);
192                 }
193             });
194         }
195         else if (tokenValue in tokenCache) {
196             winston.debug('Getting credentials from cache');
197             updateCredentials(tokenCache[tokenValue]);
198         }
199     }
200 };
```

Let's highlight the key elements of this function. First, on line **171**, you'll see that we start by extracting the token from the request. This is the token that you received from Cognito after you authenticated your user. This system then decodes the token and acquires your username on lines **174** and **175**. Next, the function makes a series of calls in sequence. It starts by looking up the user pool for the current user. It then calls **authenticateUserInPool().** This function, which is part of the TokenManager helper, ultimately calls the Cognito **getCredentialsWithIdentity()** method, passing in the token from the user. This call to Cognito triggers the role mapping we configured earlier. Cognito will extract the role from the supplied token and match it to the policy, then construct a temporary set of scoped credentials that are returned to the calling function.

Step 3 – So that's what the code is doing behind the scenes. Now, let's deploy this new version of the product manager service to see it in action.

Step 4– Our next step is to deploy version 5 of the product manager, within our Cloud9 IDE. Navigate to **Lab3/Part4/scripts** directory, right-click **product-manager-v6.sh**, and click Run to execute the shell script.



Step 5 – Wait for the **product-manager-v6.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 6 – All that remains now is to verify that all of the moving parts of this process are working. Navigate to the application using the same URL you've used throughout. If the prior user is stilled logged in, log out using the dropdown the tenant name at the top left of the application navigation bar. Now, login as **TenantOne** and access your data. Everything should work.

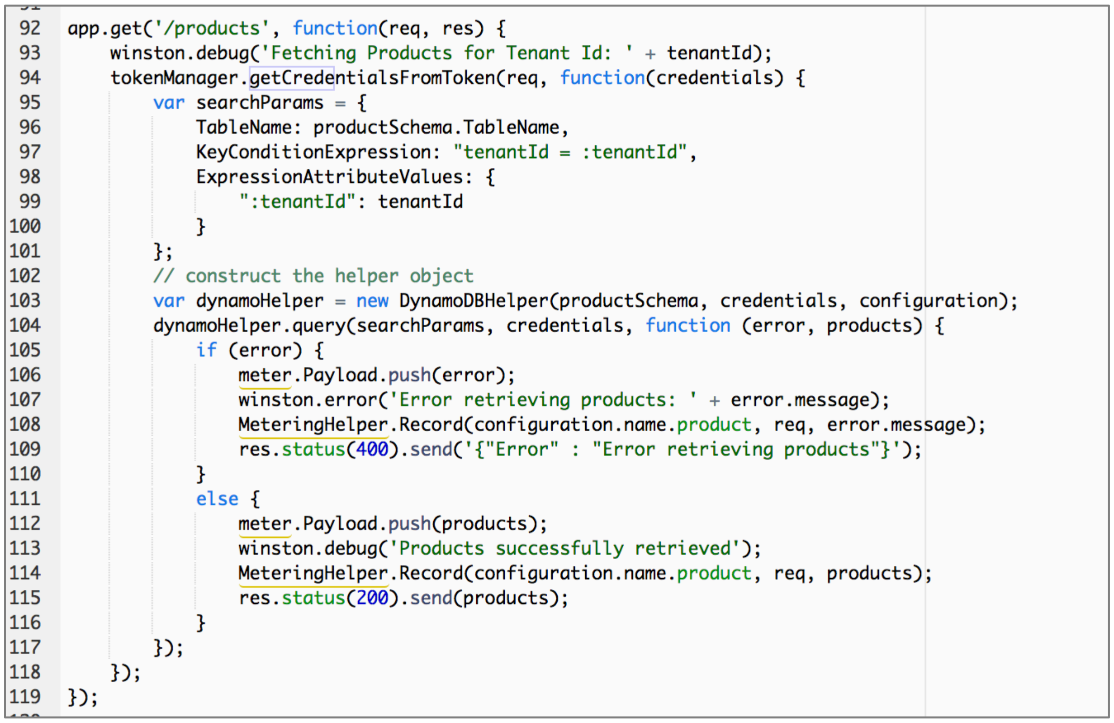
Step 7 – While seeing this work is great, it's hard to know that this new code is truly enforcing our tenant isolation. This always of tough case to test. Let's try a bit of a brute force method in Part 5.

Part 5 – Verifying Tenant-Scoped Credentials

At this point, we have incorporated security at the IAM Level by leveraging **getCredentialsWithIdentity()**, however we have not evaluated if we attempt to override the tenant identifier in the code whether this can circumvent the security incorporated in IAM. In Part 5, we will add a hardcoded Tenant ID and attempt to cross tenant boundaries defined by IAM, and we will demonstrate that so long as the access policies and roles defined previously are properly configured this is not possible by a user.

Step 1 – Let's modify the source code for our latest product manager service and manually inject a tenant identifier (similar to what was done above). In Cloud9, navigate to "**Lab3/Part5/app/source/product-manager/**", right-click **server.js** and click "Open."

Step 2 – Now let's open this **server.js** file in a text editor. Locate the line in the function where you are fetching a list of products. The code function will appear as follows:

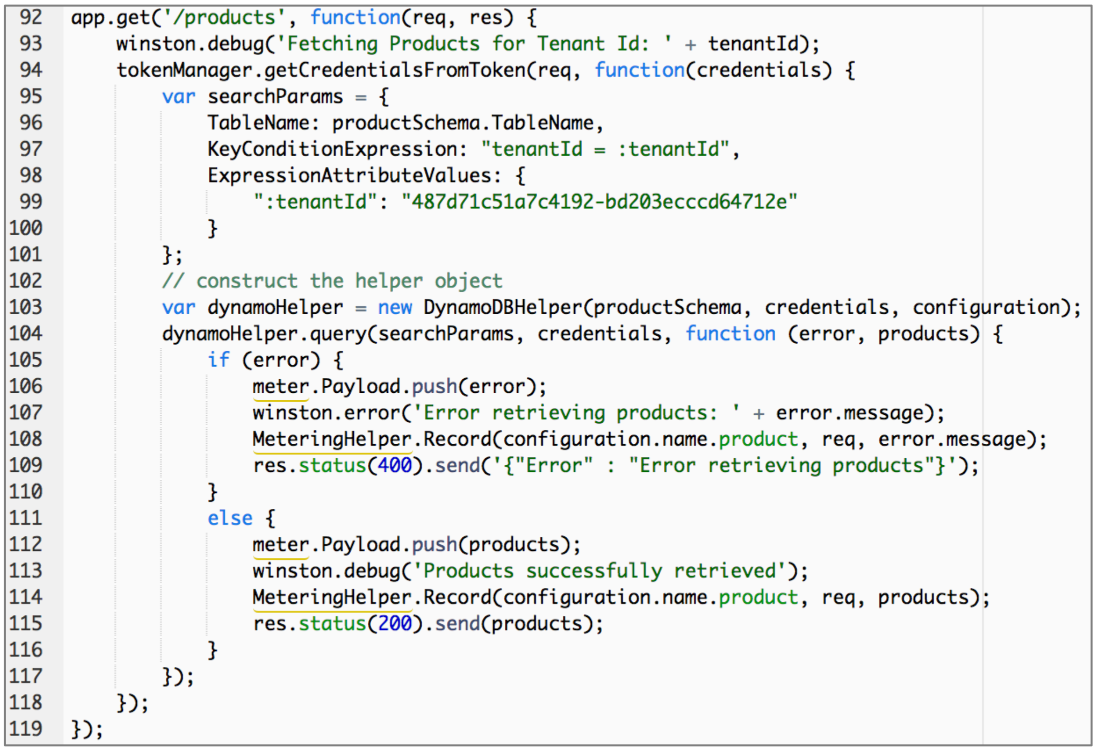


```
 92 app.get('/products', function(req, res) {
 93     winston.debug('Fetching Products for Tenant Id: ' + tenantId);
 94     tokenManager.getCredentialsFromToken(req, function(credentials) {
 95         var searchParams = {
 96             TableName: productSchema.TableName,
 97             KeyConditionExpression: "tenantId = :tenantId",
 98             ExpressionAttributeValues: {
 99                 ":tenantId": tenantId
100             }
101         };
102         // construct the helper object
103         var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
104         dynamoHelper.query(searchParams, credentials, function (error, products) {
105             if (error) {
106                 meter.Payload.push(error);
107                 winston.error('Error retrieving products: ' + error.message);
108                 MeteringHelper.Record(configuration.name.product, req, error.message);
109                 res.status(400).send('{"Error" : "Error retrieving products"}');
110             }
111             else {
112                 meter.Payload.push(products);
113                 winston.debug('Products successfully retrieved');
114                 MeteringHelper.Record(configuration.name.product, req, products);
115                 res.status(200).send(products);
116             }
117         });
118     });
119 });
```

This function is the function represents a slightly modified version of the copy we deployed at the outset of this lab. The key different here is the added tenant isolation context added at line **94**. Now, we will once again manually inject a tenant id to see how/if our new code will prevent cross tenant access. Make a changed to line **99**, replacing the value of tenantId with the hard-coded GUID of TenantTwo.

":tenantId": "<INSERT TENANTTWO GUID HERE>"

When you're done, the line should be similar to the following:



```
app.get('/products', function(req, res) {
    winston.debug('Fetching Products for Tenant Id: ' + tenantId);
    tokenManager.getCredentialsFromToken(req, function(credentials) {
        var searchParams = {
            TableName: productSchema.TableName,
            KeyConditionExpression: "tenantId = :tenantId",
            ExpressionAttributeValues: {
                ":tenantId": "487d71c51a7c4192-bd203ecccd64712e"
            }
        };
        // construct the helper object
        var dynamoHelper = new DynamoDBHelper(productSchema, credentials, configuration);
        dynamoHelper.query(searchParams, credentials, function (error, products) {
            if (error) {
                meter.Payload.push(error);
                winston.error('Error retrieving products: ' + error.message);
                MeteringHelper.Record(configuration.name.product, req, error.message);
                res.status(400).send('{"Error" : "Error retrieving products"}');
            }
            else {
                meter.Payload.push(products);
                winston.debug('Products successfully retrieved');
                MeteringHelper.Record(configuration.name.product, req, products);
                res.status(200).send(products);
            }
        });
    });
});
```

Step 3 – So that's what the code is doing behind the scenes. Now, let's deploy this new version of the product manager service to see it in action.

Step 4– Our next step is to deploy version 5 of the product manager, within our Cloud9 IDE. Navigate to **Lab3/Part5/scripts** directory, right-click **product-manager-v7.sh**, and click Run to execute the shell script.

Step 5 – With our new version of the service deployed, we can now see how this impacted the application. Let's log back into the system as with the credentials for **TenantOne** that you created above. Navigate to the URL for the application and enter your credentials (if the application is still logged in, log out using the dropdown at the top right of the page).

Step 6 – Select the "Catalog" menu option at the top of the page. This should display the catalog for your tenant. Now, you'll see that *no* products are displayed. In fact, if you look at the logs (use your browser's developer tools), you'll see that this threw and error. This is because we're logged in as **TenantOne** and our service has hard-coded **TenantTwo**. This makes it clear that our isolation policies are being enforced since the credentials we acquired prohibited us from accessing data for **TenantTwo**.

Recap: With this last step, we connected all the concepts of tenant isolation in the code of the product manager service. We added specific calls to exchange our authenticated token for a tenant-scope set of credentials, then used these credentials to access the DynamoDB table. With this new level of isolation enforcement in place, we attempted to hard-code something that crossed a tenant boundary and confirmed that our policies prohibited cross-tenant access.