This document provides a detailed explanation of the [AWS-Quickstart SaaS-Identiy-Cognito](https://github.com/aws-quickstart/saas-identity-cognito) architecture being used as consideration for a starting point for the Data Innovations proposed AWS Cloud Infrastructure. The architecture describe here has been modified. The original design included a sample order management system that has been removed. Although discussed as a matter of reference and to facilitate understanding concepts driven by the design, the AngularJS Client application has also been removed.

Table of Contents

[Reference Architecture 5](#_Toc5598639)

[Proposed DI Architecture 6](#_Toc5598640)

[Reference Architecture Breakdown 7](#_Toc5598641)

[Configuration Code 7](#_Toc5598642)

[Configuration Code Structure 7](#_Toc5598643)

[VPC Resource 7](#_Toc5598644)

[Infrastructure Resource 8](#_Toc5598645)

[SecurityGroups Resource 8](#_Toc5598646)

[ECSHostSecurityGroup Resource 8](#_Toc5598647)

[LoadBalancerSecurityGroup Resource 8](#_Toc5598648)

[ALB Resource 8](#_Toc5598649)

[LoadBalancer Resource 8](#_Toc5598650)

[LoadBalancerListener Resource 8](#_Toc5598651)

[DefaultTargetGroup Resource 8](#_Toc5598652)

[ECS Resource 8](#_Toc5598653)

[ECSCluster Resource 9](#_Toc5598654)

[ECSAutoScalingGroup Resource 9](#_Toc5598655)

[ECSLaunchConfiguration Resource 9](#_Toc5598656)

[ECSRole Resource 9](#_Toc5598657)

[ECSInstanceProfile Resource 9](#_Toc5598658)

[Role Resource 9](#_Toc5598659)

[CleanupRole Resource 9](#_Toc5598660)

[SNSRole Resource 9](#_Toc5598661)

[ApiAuthRole Resource 9](#_Toc5598662)

[LambdaExecutionRole Resource 9](#_Toc5598663)

[CloudFormationExecutionRole Resource 9](#_Toc5598664)

[API Resource 9](#_Toc5598665)

[LambdaAuthorizer Resource 10](#_Toc5598666)

[IdentityApi Resource 10](#_Toc5598667)

[ApiGatewayCloudWatchLogsRole Resource 10](#_Toc5598668)

[IdentityApiStage Resource 10](#_Toc5598669)

[ApiDeployment Resource 10](#_Toc5598670)

[WebClient Resource 10](#_Toc5598671)

[WebBucket Resource 10](#_Toc5598672)

[CloudFormationExecutionRole Resource 10](#_Toc5598673)

[CodeBuildServiceRole Resource 10](#_Toc5598674)

[CodePipelineServiceRole Resource 10](#_Toc5598675)

[ArtifactBucket Resource 11](#_Toc5598676)

[CodeBuildProject Resource 11](#_Toc5598677)

[Pipeline Resource 11](#_Toc5598678)

[WebPerm Resource 11](#_Toc5598679)

[ConfigRole Resource 12](#_Toc5598680)

[UpdateConfig Resource 12](#_Toc5598681)

[UpdateConfigFunction Resource 12](#_Toc5598682)

[BucketRepository Resource 12](#_Toc5598683)

[Repository resource 12](#_Toc5598684)

[ArtifactBucket Resource 12](#_Toc5598685)

[DynamoDBTables Resource 12](#_Toc5598686)

[UserDynamoDBTable Resource 12](#_Toc5598687)

[TenantDynamoDBTable Resource 12](#_Toc5598688)

[MICRO SERVICE RESOURCES 12](#_Toc5598689)

[CodeBuildServiceRole Resource 13](#_Toc5598690)

[CodePipelineServiceRole Resource 13](#_Toc5598691)

[CodeBuildProject Resource 13](#_Toc5598692)

[TestProject Resource 13](#_Toc5598693)

[Pipeline Resource 13](#_Toc5598694)

[CLEANUP RESOURCES 14](#_Toc5598695)

[DeleteStack Resource 14](#_Toc5598696)

[DeleteStackFunction Resource 14](#_Toc5598697)

[BucketRepositoryCleanup Resource 14](#_Toc5598698)

[BucketCleanup Resource 14](#_Toc5598699)

[RespositoryCleanup Resource 14](#_Toc5598700)

[WebClientCleanup and WebArtifactCleanup 15](#_Toc5598701)

[SysAdmin Resource 15](#_Toc5598702)

[ConfigRole Resource 15](#_Toc5598703)

[CreateSysAdmin Resource 15](#_Toc5598704)

[UpdateConfigFunction Resource 15](#_Toc5598705)

[SysAdminCleanup Resource 15](#_Toc5598706)

[ConfigRole Resource 16](#_Toc5598707)

[CleanSystem Resource 16](#_Toc5598708)

[UpdateConfigFunction Resource 16](#_Toc5598709)

[Tenant and User Provisioning 16](#_Toc5598710)

[Isolation 22](#_Toc5598711)

[Cognito 22](#_Toc5598712)

[Database Isolation 22](#_Toc5598713)

[Orchestrating Authentication 23](#_Toc5598714)

[Binding Identity to Tenant and Policies 24](#_Toc5598715)

[Tokens 24](#_Toc5598716)

[API Gateway and Custom Lambda Authorizer 24](#_Toc5598717)

[Recommended Modifications 26](#_Toc5598718)

[NodeJs code 26](#_Toc5598719)

[Python code 27](#_Toc5598720)

[API Gateway 27](#_Toc5598721)

[AWS build tools 27](#_Toc5598722)

[Tenant data isolation 27](#_Toc5598723)

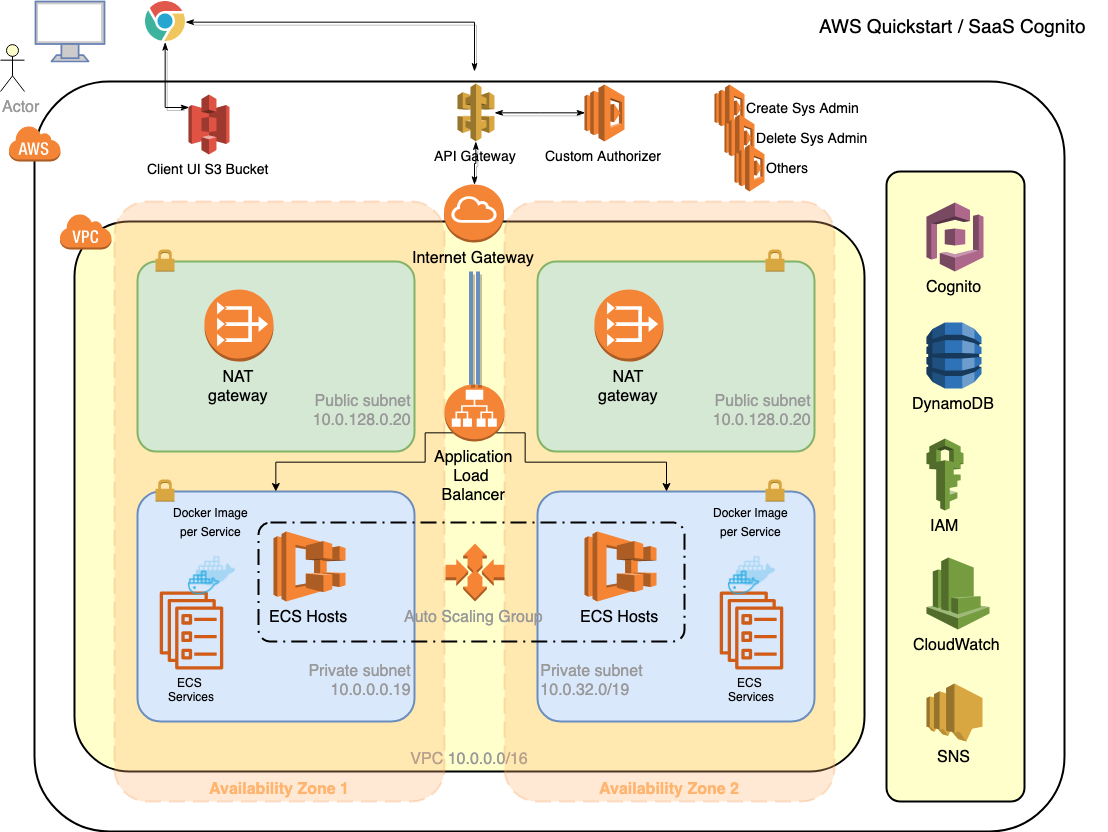
[Roles 28](#_Toc5598724)

[Inter-process communications 28](#_Toc5598725)

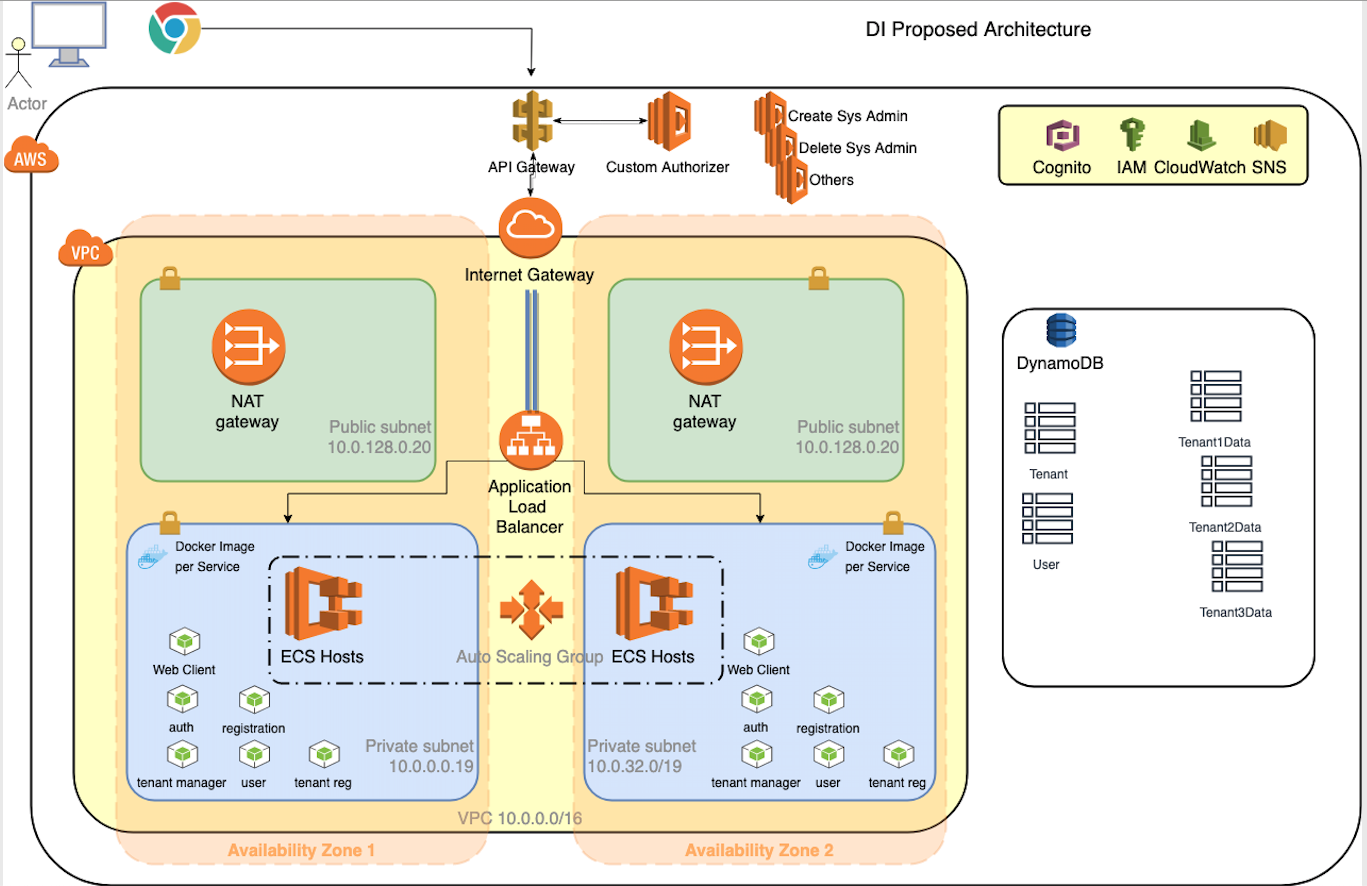
[System and Code lockdown 28](#_Toc5598726)

[Considerations for Production Environments from the Quick Start document 28](#_Toc5598727)

# Reference Architecture



# Proposed DI Architecture



# Reference Architecture Breakdown

## Configuration Code

* + The initial quick start link for the reference architecture references code in the Git repo at: <https://github.com/aws-quickstart/saas-identity-cognito>
  + The initial cloud formation to launch the install referenced code that wasn’t accessible, so I’ve pulled it into a project on my local machine and will ultimately be pushed to DI Git
  + The code has been modified to remove the order entry aspects as well as modifications to the included UI removing the order entry aspects.
  + This document won’t serve as instructions for installing the configuration code but will document the Resources built by the configuration.

## Configuration Code Structure

* + There is an initial file that is launched via the console titled ***ref-arch-start.yml.***
  + This file creates 2 **Resources** the first being the VPC shown in the architecture diagram. The second resource is a nested Cloud Formation Stack of nested CF Stacks that builds out the rest of the infrastructure.

### VPC Resource

* Creation of 2 public subnets ***Public subnet 1*** and ***Public subnet 2***.
  + Each subnet is placed in one of 2 unique Availability Zones (AZ) with the region being deployed to that were chosen in the console interface.
  + Subnets use CIDRs of 10.0.128.0/20 and 10.0.144.0/20 providing 4089 IPs each
* Creation of 2 private subnets ***Private subnet 1A*** and ***Private subnet 2A.***
  + Each subnet is placed in one of 2 unique AZs with the region being deployed to that were chosen in the console interface.
  + Subnets use CIDRs of 10.0.0.0/19 and 10.0.32.0/19 for 8185 IPs each.
* Creation of an Internet Gateway (IGW) that provide external access to the 2 public subnets.
  + Both public subnet routing tables point to 0.0.0.0/0 with the IGW as the target.
* Creation of 2 NAT Gateways (NG) that provide high availability routing of traffic that flows from your private subnets to other AWS services or to the Internet.
* Creation of routing tables for all subnets.
* Creation of 2 Elastic IPs (EIPs) assigned to each of the NGs.
* Creation of an Endpoint for connections from the private subnets to S3

### Infrastructure Resource

* Outputs from the previous stack are used as input Parameters to this resource which a Cloud Formation Stack (CFS)
* This stack is a nested one using the file ***ref-arch-infrastructure.yml.*** It builds out 36 resources. 10 of these are merely Wait Handles and Wait Handles Conditions which are devices used to coordinate stack resource creation with configuration actions that are external to the stack creation and to track the status of a configuration process. Those 10 will be omitted here.

#### SecurityGroups Resource

Creates 2 Security Groups using ***ref-arch-security-groups.yml***

##### ECSHostSecurityGroup Resource

* + - Creates SecurityGroup which allows access on all ports to ECS hosts, tasks, and containers for the Load Balancer

##### LoadBalancerSecurityGroup Resource

* + - Creates SecurityGroup that allows traffic on all ports. Will be applied to the Application Load Balancer

#### ALB Resource

Creates an Application Load Balancer using ***ref-arch-load-balancers.yml*** and applying the LoadBalancerSecurityGroup

##### LoadBalancer Resource

* + - Creates an Application Load Balancer directing and balancing traffic to the 2 Public Subnets.

##### LoadBalancerListener Resource

* + - Creates a Listener process that checks for connections requests via HTTP on Port 80 and forwards them to the DefaultTargetGroup

##### DefaultTargetGroup Resource

* + - Creates a TargetGroup that routes requests to registered targets which will be the ECS Clusters

#### ECS Resource

Creates an ECS cluster in an Auto Scaling Group using the previously created VPC and subnets. Uses ***ref-arch-ecs-cluster.yml***.

##### ECSCluster Resource

* + - Defines and ECS cluster using the StackName for name

##### ECSAutoScalingGroup Resource

* + - Creates an Auto Scaling Group with passed in values for MinSize and MaxSize and DesiredCapacity.
    - Defines a CreationPolicy and an UpdatePolicy

##### ECSLaunchConfiguration Resource

* + - Creates a Launch Configuration that is used by the Auto Scaling Group to configure EC2 Instances.

##### ECSRole Resource

* + - Creates Role for EC2 service access to ECS, Logging, DynamoDb, IAM, Cognito Identity Pools, Cognito User Pools, and APIGW Certificates.

##### ECSInstanceProfile Resource

* + - Creates an Instance Profile that can be used with IAM Roles for EC2 Instances based on ECSRole

#### Role Resource

Creates IAM Roles the system components will need to interact. Uses ***ref-arch-roles.yml***

##### CleanupRole Resource

* + - This is a very broad role allowing complete access to resources belonging to Logging, ECR, ECS, EC2, ELB, CF, S3, and IAM and is granted to the Lambda service.
    - This role is used in the <X>Cleanup resources in ***ref-arch-infrastructure.yml*** that typically clean up out of band resources created in the stack

##### SNSRole Resource

* + - Creates a Role granted to Cognito that allows Publish access to all Topics

##### ApiAuthRole Resource

* + - Creates a Role granted to APIGW to invoke Lambda Functions

##### LambdaExecutionRole Resource

* + - Creates a Role granted to Lambda that allows Logging access

##### CloudFormationExecutionRole Resource

* + - Creates a Role granted to CF that allows broad access to ECS, ECR, IAM, S3, EC2, ELB, and Logging

#### API Resource

Creates an API Gateway. Uses ***ref-arch-api-gateway.yml***.

##### LambdaAuthorizer Resource

* + - Creates a Lambda function that will be the Custom Authorizer for APIGW
    - Code is in ***functions/packages/custom-authorizer/custom-authorizer.zip***

##### IdentityApi Resource

* + - Creates an APIGW RestApi.
    - Uses an inline Swagger 2.0 Spec to define the paths

##### ApiGatewayCloudWatchLogsRole Resource

* + - Creates a role granted to APIGW that allows access to Logging

##### IdentityApiStage Resource

* + - Creates an ApiGateway Stage named ***prod*** for the IdentityApi resource using the ApiDeployment resource for deployment.

##### ApiDeployment Resource

* + - Creates and ApiGateway Deployment that instantiates the prod Stage of the IdentityApi.

#### WebClient Resource

Creates a CF Stack to build a Deployment Pipeline for the Web Client. Uses ***ref-arch-web-deployment-pipeline.yml***

##### WebBucket Resource

* + - Creates an S3 with Public Read access and Configured as a Web Site with CORS enabled.
    - It allows all headers and only supports GET commands.
    - It expects a file ***index.html*** as the Index Document and a file ***error.html*** as it’s Error Document.

##### CloudFormationExecutionRole Resource

* + - Creates a Role granted to CF that allows access to ECS, ECR, IAM, S3, EC2, ELB, and Logs

##### CodeBuildServiceRole Resource

* + - Creates a role granted to CodeBuild that grants access to certain Logging actions, full access to bucket created in WebBucket resource, and full access to the bucket created in ArtifactBucket resource.

##### CodePipelineServiceRole Resource

* + - Creates a Role granted to CodePipeline that recursive access to the bucket created in ArtifactBucket resource, the bucket created in WebBucket resource, and the TemplateBucket passed into this CF template.
    - It also grants access to CF, IAM, EC2, ELB, Logging, and CodeBuild

##### ArtifactBucket Resource

* + - Creates an un-named Bucket used above

##### CodeBuildProject Resource

* + - Creates a CodeBuild Project.
    - Specifies a NodeJS v8.11.0 build Environment on a Small Linux build machine
    - Uses the CodeBuildServiceRole defined previously
    - Phases
      * Pre\_build
        + Installs components required by the Web Client for building
      * Build
        + Uses Grunt to build the project
      * Post\_build
        + Runs AWS Cli command to copy the build artifact to an S3 bucket specified in it’s environment which is the WebBucket resource created earlier.
        + Runs a ***curl*** command that signals the Wait Handle

##### Pipeline Resource

* + - Creates a CodePipeline Pipeline that will use the CodeBuildProject resource to build the Web Client UI application
    - Uses the CodePipelineServiceRole created earlier
    - Stages
      * Source
        + Defines a single Action named App that defines the S3 bucket where Web Client zipped code resides
      * Build
        + Defines a single Action named Build that uses the CodeBuildProject resource to build the App source

#### WebPerm Resource

Creates a CF Stack that correctly sets all of the permissions on the files in the WebBucket resource. This ensures the Web Client is readable for the world. Uses ***ref-arch-bucket-public-read.yml***.

##### ConfigRole Resource

* + - Creates a Role granted to Lambda that allows certain Logging actions and access to all contents of the WebBucket on S3

##### UpdateConfig Resource

* + - Creates a Custom ConfigFile that is a custom resource. It’s a pattern to allow CF to trigger Lambda functions created within the Stack.
    - Its ServiceToken is set to the ARN for the UpdateConfigFunction resource

##### UpdateConfigFunction Resource

* + - Creates a Lambda Function with inline Python code that will update the permissions on the WebBucket

#### BucketRepository Resource

Creates an ECR and a bucket for Docker images. Uses ***ref-arch-docker-bucket-repository.yml***

##### Repository resource

* + - Creates a non-named ECR Repository
    - It’s important that the repository is unnamed. If you specify a name, you cannot perform updates that require replacement of this resource. You can perform updates that require no or some interruption. If you must replace the resource, specify a new name.

##### ArtifactBucket Resource

* + - Creates an S3 Bucket that is version enabled and has a Detain retention policy. Buckets used for ECR must be set up this way.

#### DynamoDBTables Resource

Creates the DynamoDB Tables needed to support Users and Tenants. Uses ***ref-arch-dynamodb-tables.yml***

##### UserDynamoDBTable Resource

* + - Creates a table with the passed in name of UserTable.
    - Sets Attributes, KeySchema, Provisioned Throughput, and Global Secondary Indices.

##### TenantDynamoDBTable Resource

* + - Creates a table with the passed in anme of TenantTable.
    - Sets Attributes, KeySchema and Provisioned Throuput.

#### MICRO SERVICE RESOURCES

There are currently 5 micro service resources: UserService, AuthService, TenanService, RegService, and SysService. The resouces are CF Stack resources built using the same nested stack file ***ref-arch-deployment-pipeline-test.yml***.

The resources are built in order with each one dependent upon the completion of the previous. The order doesn’t imply anything other than the need to create them one at a time.

##### CodeBuildServiceRole Resource

* + - Creates a Role granted to CodeBuild that provides access to Logging, certain actions on the ArtifactBucket Resource in S3, and various actions in ECR on the Repository resource.

##### CodePipelineServiceRole Resource

* + - Creates a Role granted to CodePipeline that provides access to Loggging, the ArtifactBucket and TemplateBucket resources in S3, CodeBuild, CF, IAM, EC2, and ELB

##### CodeBuildProject Resource

* + - Creates a CodeBuild Project that logs into ECR, performs a Docker build from the supplied Source in S3 Bucket tagging it with a value passed in, and pushes the image to Repository passed in.

##### TestProject Resource

* + - Creates a CodeBuild Project that tests the deployment of the Pipeline resource by pinging the Healthcheck URL via a curl command.
    - It then notifies the Wait Handle via another curl command to the passed in Wait Handle URL.

##### Pipeline Resource

* + - Creates a CodePipeline Pipeline that runs 4 Stages
      * Source Stage
        + Detects changes to the micro service application source that is stored in the S3 bucket and pulls them into the pipeline.
        + Detects changes to the service.template.zip file source that is stored in the S3 bucket and pulls them into the pipeline. This file contains a CF Template that spins up an ECS Service, Task Definition, Target Group, and ALB Path for the micro service being built.
      * Build Stage
        + Uses the CodeBuildProject resource as it’s configuration to run a Build Category. This will build the micro service code and output an artifact called BuildOutput
      * Deploy Stage
        + This is a CF stage that will use the service.template and a config.template to spin up the resources mentioned in the Source stage while deploying the micro service to the instances.
      * Test Stage
        + This is a CodeBuild stage that will build the TestProject resource. This will essentially test the health of the Deploy stage and indicate to the parent stack completion.

#### CLEANUP RESOURCES

There are currently 5 cleanup related resources: UserServiceCleanup, AuthServiceCleanup, TenantServiceCleanup, RegServiceCleanup, and SysServiceCleanup. The resouces are CF Stack resources built using the same nested stack file ***ref-arch-clean-cfnstack.yml***.

##### DeleteStack Resource

* + - Creates a Custom Resource using the ARN of the DeleteStackFunction Lambda resource. When a lifecycle method of the Stack passed in as a parameter changes, the Lambda resource is triggered.

##### DeleteStackFunction Resource

* + - Creates a Lambda function that is activated via the DeleteStack Custom resource. Only handles he Delete lifecycle method in a meaningful way by deleting the stack.

##### BucketRepositoryCleanup Resource

Creates a CF Stack to create resources for deleting the ArtifactBucket resource and the Repository resource. Uses the nested file ***ref-arch-cleanup-bucket-repository.yml***.

##### BucketCleanup Resource

Creates a CF Stack that deletes a Bucket. Uses file ***ref-arch-clean-bucket.yml***.

* + - * UpdateConfig Resource
        + Creates a Custom Resource triggered by the ARN for the UpdateConfigFunction resource.
        + Passes in the Bucket to delete and the Region for the Bucket.
      * UpdateConfigFunction Resource
        + Creates a Lambda Function that handles primarily the Delete event by recursively emptying and then deleting the Bucket supplied.

##### RespositoryCleanup Resource

Creates a CF Stack that will delete the passed in Repository resource. Uses file ***ref-arch-clean-repository.yml***.

* + - * UpdateConfig Resource
        + Creates a Custom Resource triggered by the ARN for the UpdateConfigFunction resource.
        + Passes in the Repository to be removed.
      * UpdateConfigFunction Resource

Creates a Lambda Function that handles primarily the Delete event by deleting the Repository supplied.

##### WebClientCleanup and WebArtifactCleanup

Creates a CF Stack that deletes a Bucket. Uses file ***ref-arch-clean-bucket.yml***.

* + - UpdateConfig Resource
      * Creates a Custom Resource triggered by the ARN for the UpdateConfigFunction resource.
      * Passes in the Bucket to delete and the Region for the Bucket.
    - UpdateConfigFunction Resource
      * Creates a Lambda Function that handles primarily the Delete event by recursively emptying and then deleting the Bucket supplied.

#### SysAdmin Resource

Creates a CF Stack that creates an initial system admin for the entire multi-tenant system. Uses file ***ref-arch-rest-request-create.yml***.

##### ConfigRole Resource

* + - Creates a Role granted to Lambda that provides access to Logging that will be assumed by the UpdateConfigFunction resource

##### CreateSysAdmin Resource

* + - Creates a Custom Resource triggered by the ARN for the UpdateConfigFunction.
    - Passes in the Domain, Protocol, Path, Method, and Body

##### UpdateConfigFunction Resource

* + - Creates a Lambda Function that uses the code at ***/functions/packages/rest-request-create/rest-request-create.zip***
    - The code will make an REST call to SysService at /sys/admin POSTing the Body parameter passed into the call. This will trigger the creation of the intial Sys Admin user for the whole system. The REST end point will also send an email to the user.

#### SysAdminCleanup Resource

Creates a CF Stack that cleans up resources not required for the initial system admin created in the SysAdmin resource. Uses file ***ref-arch-rest-request-delete.yml***.

##### ConfigRole Resource

* + - Creates a Role granted to Lambda that provides access to Logging that will be assumed by the UpdateConfigFunction resource

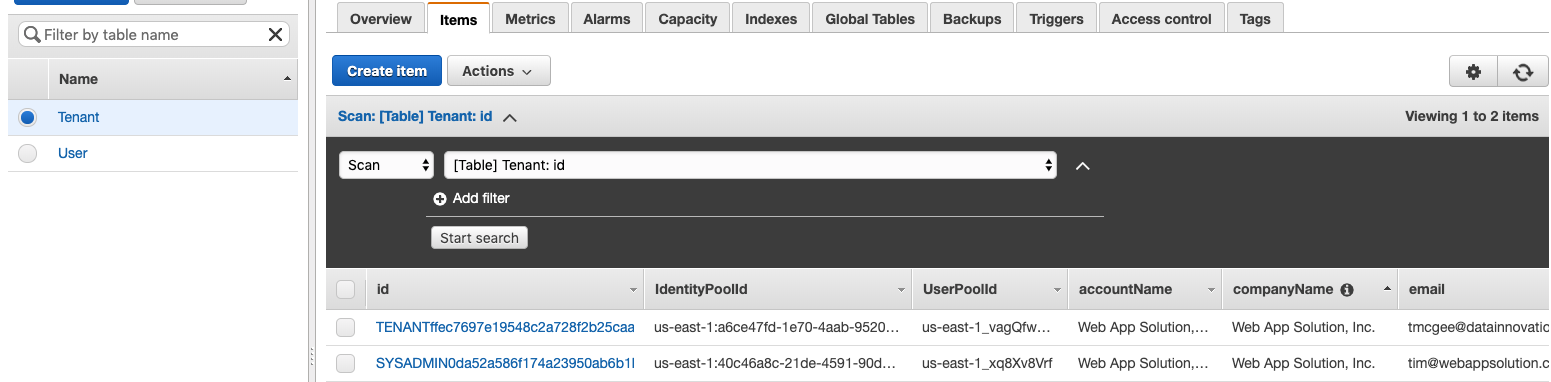
##### CleanSystem Resource

* + - Creates a Custom Resource triggered by the ARN for the UpdateConfigFunction.
    - Passes in the Domain, Protocol, Path, Method, and Body

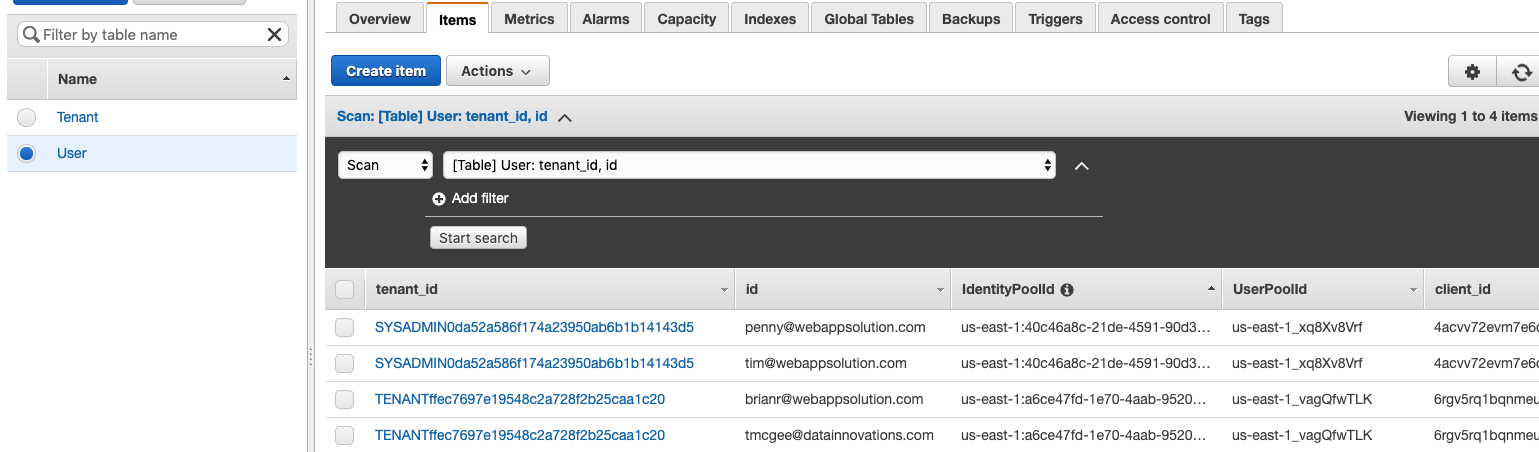
##### UpdateConfigFunction Resource

* + - Creates a Lambda Function that uses the code at ***/functions/packages/rest-request-delete/rest-request-create.zip***
    - The code will make a DELETE REST call to UserService at /user/tenants with the Body parameter passed into the call. This will trigger the deletion of the Sys Admin as a Tenant.

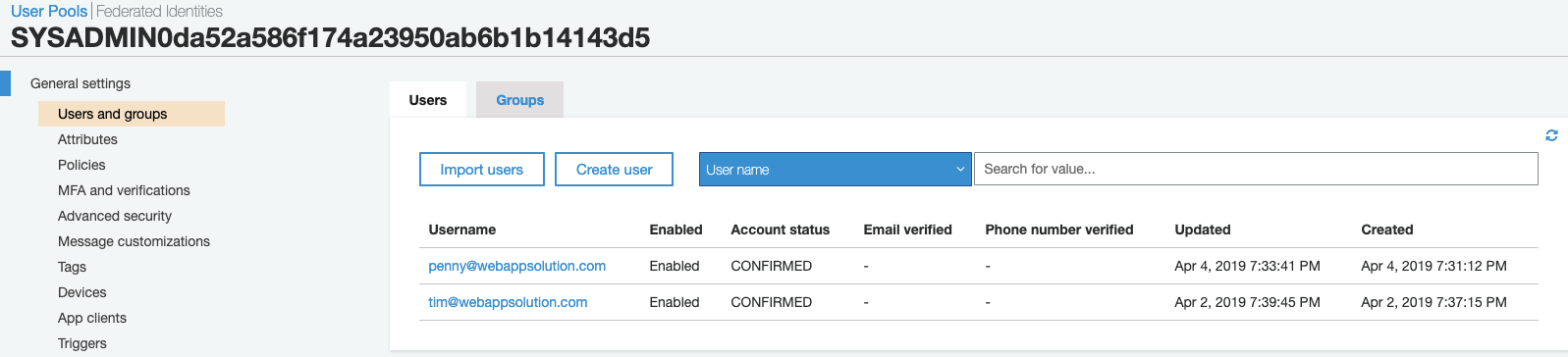
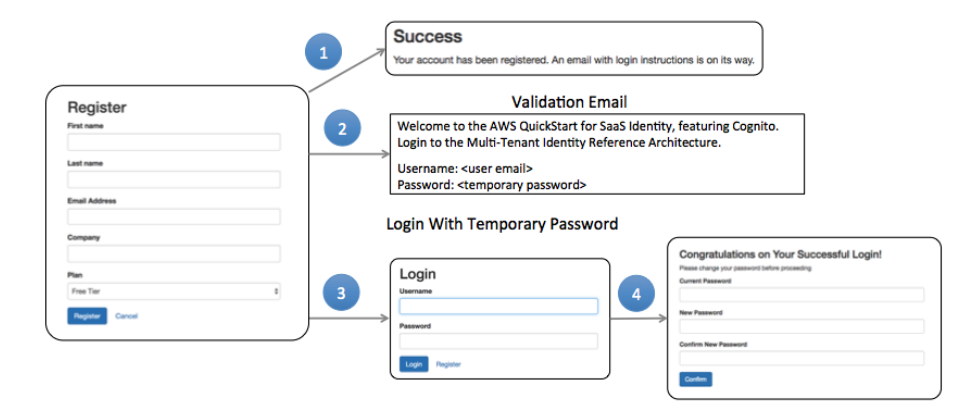
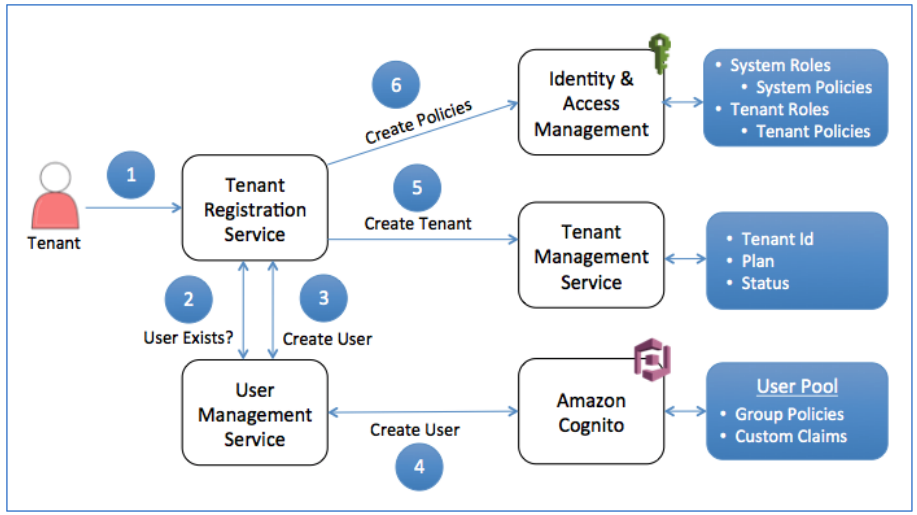
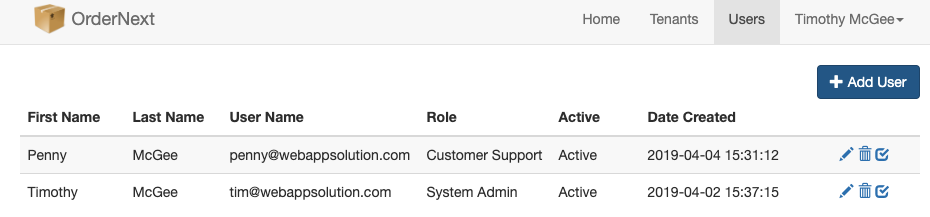
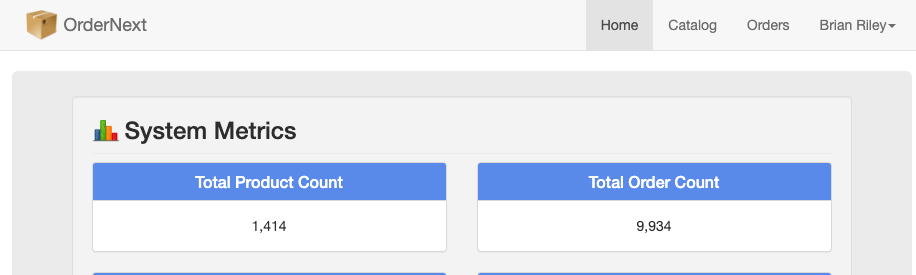
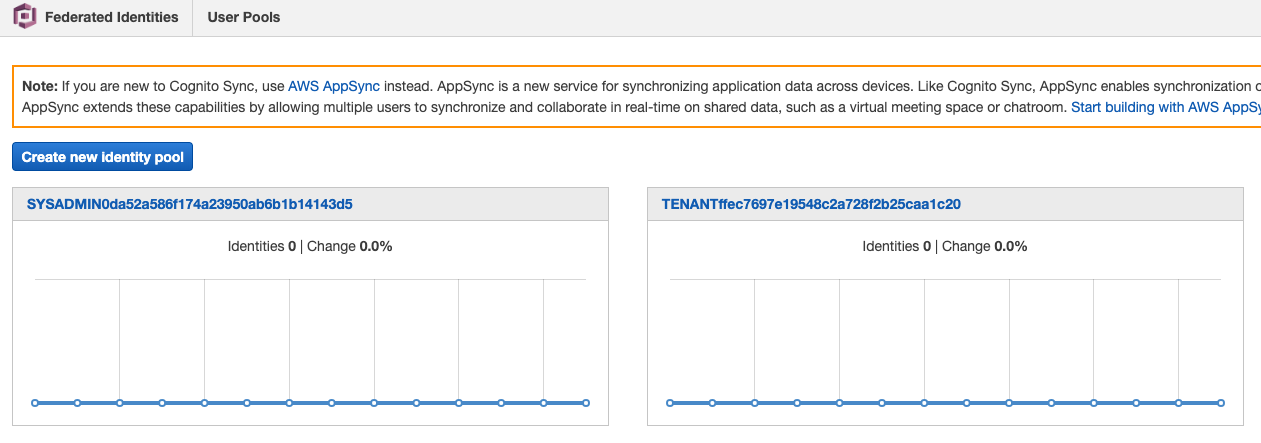
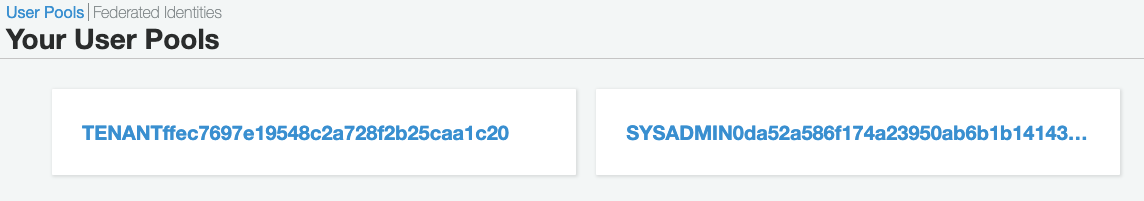
#### Tenant and User Provisioning

* + The script ***ref-arch-start.yml*** presents a user interface in the console that asks the user to enter certain values. There are fields the user will input to identify the initial System Admin created when the scripts complete. A row will be created in both the Tenant and User tables. Highlighted in yellow is the row created in each table using [tim@webappsolution.com](mailto:tim@webappsolution.com) for the initial email address. The CF Stack ***ref-arch-rest-request-create.yml*** utllizes a Lambda function to make a ReST call to the ***system-registration Service*** which has deployed as a Docker image in the ECS Cluster. This service makes ReST calls to other services in the stack to facilitate provisioning.   
      
      
    







* + You’ll notice that the Users and Tenants are associated with a Cognito Identity Pool Id and a Cognito User Pool Id. There are no Roles defined in the Identity Pool. This is where Roles such as SysAdmin, Customer Support, etc would be defined and applied to Users. (Omitting image of Identity Pool since it lacks Roles there’s not much to see.)  
      
      
    
  + In the Tenant table, you’ll notice there is another entry. This points to [tmcgee@datainnovations.com](mailto:tmcgee@datainnovations.com). This Tenant was created in the Client app by going to the Login page and selecting Register. The new Tenant would fill in their information and be provisioned in the system. The Client app makes ReST calls to the ***tenant-registration Service*** which has deployed as a Docker image in the ECS Cluster. This service makes ReST calls to other services in the stack to facilitate provisioning. The following image outlines the Tenant onboarding flow in the Client application.  
      
      
      
      
    
  + The following diagram provides a highly conceptualized version of the onboarding process as it interacts with the system components.  
      
    
  + The steps are
    - New Tenant completes and submits the registration form
    - Registration confirms the user doesn’t already exist
    - User Management creates a new user in DynamoDB
    - User management creates a new User Pool and Custom Claims in Cognito
    - Tenant Manager creates new Tenant in DynamoDB
    - Tenant Registration provisions policies for each Tenant Role
  + In the User table, you’ll note there are 3 additional entries besides the initial System Administrator.
  + [penny@webappsolution.com](mailto:penny@webappsolution.com) was added as another SysAdmin in the primary Tenant which would represent DI. This was accomplished by being logged in as primary SysAdmin [tim@webappsolution.com](mailto:tim@webappsolution.com) and adding a user.   
      
    
  + The entry for [brianr@webappsolution.com](mailto:brianr@webappsolution.com) was added by being logged in as Tenant [tmcgee@datainnovations.com](mailto:tmcgee@datainnovations.com) and adding a User. Note that this user has different Menu choices.  
      
    
  + Once the SysAdmin added the Tenant [tmcgee@datainnovations.com](mailto:tmcgee@datainnovations.com), there was a new Cognito Identity Pool and a new Cognito User Pool created for the Tenant.  
      
      
      
    

#### Isolation

* + The Reference Architecture provides Tenant Isolation through 2 primary mechanisms

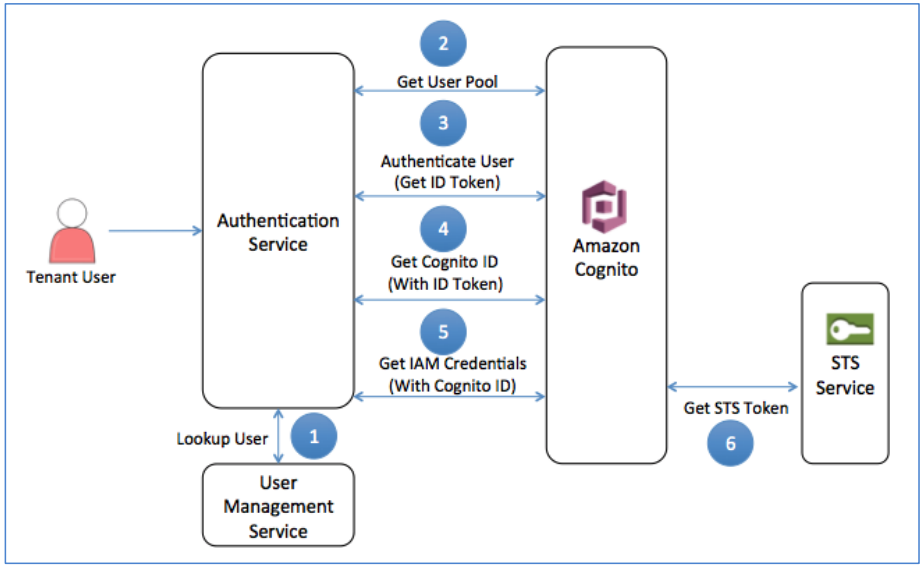
##### Cognito

* + - As shown in the images prior, there are Cognito User and Identity Pools created for the Saas IVP, in this case DI, as well as for each new Tenant created.
    - System Users (DI)
      * They require access to all Tenants. There are currently 2 variants defined, System Administrator (SA) and System User (SU). The primary difference is the ability for the SA to manage the System Users by adding, editing and deleting those Users.
    - Tenant Users
      * They represent individual Tenant roles that are used to manage Tenant configuration and assume various domain specific roles of an application. The initial Tenant User, referred to as the Owner, is created during the Registration process described above. There are currently 2 variants defined, Tenant Administrator (TA) and Tenant User (TU). The primary difference is the ability for the TA to manage Tenant Users by adding, editing, and deleting those Users.

##### Database Isolation

* + - All Tenants share the same Order and Product Tables. Data isolation is provided by using the tenantId field as the primary partition key. This value is made available via the login process and the JWT access tokens and that is provided on all calls the Client application makes to the API Gateway. The value eventually makes its way through to the underlying ***order-manager Service*** and ***product-manager Service*** and is used in queries of the underlying data.

#### Orchestrating Authentication

* + The authentication process connects Users to Tenants and gets the Context that controls Users’ access to Resources  
      
    
  + The reference architecture uses Cognitos ***Enhanced Flow***. The authentication flow is orchestrated by the systems ***auth-manager Service*** consisting of these steps
    - Authentication determines if User exists and if so determines their assigned User Pool ID stored in User table
    - Authentication retrieves information about the User Pool from Cognito. At this stage, the service has the information it needs to authenticate the user against a specific User Pool.
    - The User is authenticated against their assigned User Pool and successful authentication returns an ID token.
    - The ID token is used to get a Cognito ID from Cognito
    - Cognito receives a request for a temporary STS token from the STS Service, exchanges the supplied Cognito ID for an STS token and returns that back to the authentication service.
    - The authentication service returens the ID token and the STS token (uses as and access token) back to the Client. The ID token supplies the Client with access to claims data about the user. Both the ID and STS tokens are passed through in calls to downstream servicing. They supply both the Tenant context for the user and the Access ID and Secret keys that will be used to access AWS resources.

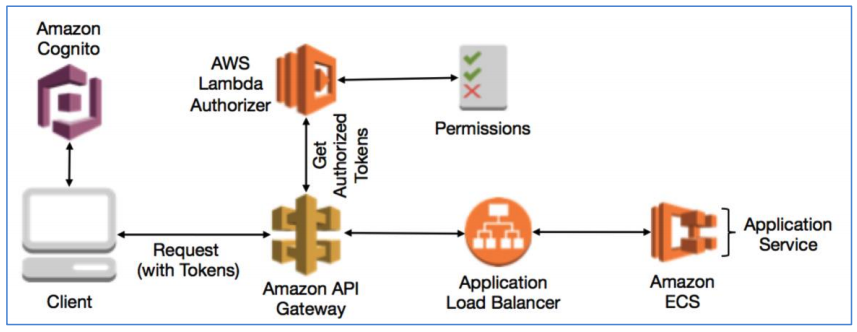
#### Binding Identity to Tenant and Policies

* + The Cognito User Pool defines Custom Attributes that are assigned to each User in the Pool. These Attributes describe the User relationship to Tenants.
    - One Custom Attribute is ***tenant\_id***which links the User to the Tenant.
    - Another is ***role*** which provides a link to IAM Policies we want this User to observe.
  + Claims are used to represent attributes that associate tenant information with each user. These claims are packaged and transported in an encoded JSON Web Token (JWT). These tokens contain data that is pulled from the attributes defined in the Amazon Cognito User Pool for a Tenant as outlined above. Cognito supports a standard set of attributes that are supported by all implementations of the OpenID Connect (OIDC) protocol. However, these standard attributes don’t include some of the items you need to associate a user with a tenant. These additional data elements must be introduced as custom attributes, which are then conveyed as custom claims in the JWT returned by the authentication process.
  + The custom attributes are provisioned and configured when each tenant is onboarded to the system. Tenant Service first creates the tenant’s user pool, and then adds the required attributes to this pool. The registration process collects and generates the data for these attributes and injects them into the user provisioning process. Then, when the user is authenticated, these attributes are encoded and returned as an ID token.

#### Tokens

* + As discussed previously, a successful authentication will yield and ID token and and Access token returned to the Client as JWTs. The Client will insert the tokens into the header of each HTTPS request to the ReST services. Each service consults the ID token to acquire Tenant context. The tokens flow through the service processes to provide the tenant context required for scoping access along with the credentials that are scoped by IAM policies.

#### API Gateway and Custom Lambda Authorizer

* + Each request from the Client is routed through the API GW which then passes them through to the correct micro service. In this regard, the API GW is acting as a reverse-proxy to the underlying services. This provides an alternative to using NGINX in the micro services hence removing another piece of software to monitor and configure across each micro service. The API GW is highly available and instantly scalable and can be presented in Edge locations where necessary.
  + Each request is passed through the Custom Lambda Authorizer to authorize access to the requested end point. The following diagram shows the high-level flow.  
      
      
      
    
  + The Lambda Function downloads the PEM Certificates required for the Cognito User Pool. These were used to sign the JWT. It then decodes the claims in the token by verifying the issuer, the type of token, and the signature of the provided ID token. If all is good, it issues approvals for the requested resource methods.

# Recommended Modifications

## NodeJs code

* + - Current micro services are built with Node version 7.1.0. The current Long Term Support version is 10.15.3 and the most current version is 11.13.0. Both support up through ES2018 (ES7). ES2017 introduced support for ***async and await***. This provides a more readable and testable approach to the nesting of ***Promise*** syntax. Since this is a new project, consideration should be given to starting out on the latest version of all code and library choices.
    - See <https://node.green/> for the latest NodeJs/JS mappings.
    - The current code uses ***express*** web application framework for its ReST implementation. The team has expressed a desire to switch to either ***Fastify*** or ***Restify*** due to performance gains over ***express*.** I have no opinion on the choice as any of the 3 are up to the task. I’ll point out that this will require modifications to the existing micro services if a move from ***express*** is chosen. This may be a non-trivial level of effort.
    - The current micro service code is lacking in some of the best practices for code writing.
      * There is not testing of the code in any form. Although much of the code will be tied to actions that aren’t easily accessible to unit testing, as much effort as possible should be taken to make the code base as functional based as possible so that a reasonable level of coverage can be made. Testing should be automated and part of the CI/CD path.
      * Static tools such as linters and formatters are not employed.
      * Documentation is minimal. Consideration of something like ***JSDoc*** might be considered if off-line text- based documentation is required.
      * API documentation of the form of Swagger is not used. This would allow a testing harness and full documentation of the API for consumers of the API.
      * There’s lack of consistent coding style throughout. This is very important given that most of the development team are new to NodeJs. Consistent and repeatable reference templates should be built for the various coding patterns that will be used in development. Adherence to these patterns should be assessed during Code Reviews.

## Python code

* + - All Python code should be migrated to NodeJs. The odd inclusion of it provides no particular usefulness and is not in the best interest homogeneity of the code base. It would also require another skill set for developers that may not have Python experience.

## API Gateway

* + - The API GW constructed in the reference architecture serves as the proxy to the underlying micro services. Calls are routed to the Application Load Balancer which then routes to the appropriate Service.
    - Use of the API GW in this ‘pass-through’ approach provides a highly available and instantly scalable solution to routing.
    - It reduces the complexity of solutions utilizing NGINX at the EC2 layer.
    - Generating all API documentation in Swagger is much less painful with the use of the ***Serverless Framework*** for scaffolding the API GW endpoints.
    - The existing Authentication mechanism of the Custom Lambda Authorizer integrates seamlessly with API GW. Removal of the API GW would require additional effort and complexity to migrate the whole strategy to a Docker Service when the cost for the current solution is nearly free.

## AWS build tools

* + - The reference architecture uses CodeBuild and CodePipeline to build and deploy the micro services as well as the web Client. These tools take advantage of their cohesion to the AWS eco-system to simplify CI/CD.
    - Efforts should be made to utilize these tools whenever possible. An on-prem Git/Jenkins solution can be used to house source code and creates builds and then deliver those to a source such as S3 for final integration into the ECR and ECS clusters.

## Tenant data isolation

* + - The current architecture supports data isolation via primary key access within tables shared between all Tenants.
    - Assuming DI should retain access to Tenants and Users at a minimum, it’s reasonable to proceed with the shared table approach for the User and Tenant tables.
    - Changes should be made to support unique tables for all Tenant specific data other than Users.
    - This will require considerable changes to the on-boarding and provisioning processes.

## Roles

* + - The current architecture supports a limited set of Roles. At both the System and Tenant levels, there are essentially Admin and User Roles.
    - A basic set of anticipated Roles should be determined very early so they can be accounted for in the on-boarding and provisioning processes including inclusion into Cognito Identity Pools.

## Inter-process communications

* + - The current reference architecture relies on ReST calls between micro services that need to coordinate flows. This should be evaluated early on and consideration given to the use of routing the calls through the API GW or using SNS and SQS where appropriate to better scale the system. DI will need to assess any concerns for ReST processing vs asynchronous processing.

## System and Code lockdown

* + - There has been discussion concerning the level of lock-down of the EC2 instances and whether any modifications to those need to be change managed.
    - The validity of this potential requirement should be considered a high priority as it may affect infrastructure build.
    - In a similar fashion, there has been discussion on the level of lock down in the developer code environment. This surrounds the use of publicly accessible libraries via mechanisms such as ***NPM***.
    - The validity of this potential requirement should be considered a high priority as it will affect many of the processes in establishing coding standards developers will adhere to. It may also require the building of an internal repository and process for managing change.

## Considerations for Production Environments from the Quick Start document

* + - Auto Scaling
      * There are currently no Policies attached to the Auto Scaling model that measure and respond to load changes on the Services. Although not an immediate concern, DI will need to perform analysis and testing of potential loads that can be used in tuning this area.
    - JWT Encoding
      * JWTs aren’t currently encoded. It’s recommended to use an encryption library to encode the Tokens to better the secure the content of them.

Proposed DI Architecture

