The Bootcamp Environment

We have run a set of CloudFormation stacks in your provisioned environment to support the different lab activities. This environment includes a VPC with multiple subnets, a number of security settings, a couple of S3 buckets and an Elastic Container Service setup that we'll use to run our microservices.

Lab 1 – Tenant Onboarding

Overview

For this first lab, we're going to look at what it takes to get tenants onboarded to a SaaS system. In many respects, onboarding is one most foundational aspects of SaaS. It creates the footprint of how tenants and users will be represented in our system, determining how tenants will be identified and conveyed as they flow through all the moving parts of our SaaS architecture.

A key part of this process is to introduce the notion of a "SaaS Identity". In many systems, the user's identity will often be represented in an identity provider completely separate from its relationship to a tenant. Tenant is then somehow resolved as separate step. This adds overhead and complexity. Instead, in this lab, you'll see how we created a tighter binding between users and tenants that creates a SaaS identity that we can then flow through the various services of our system. This allows tenant context to be promoted to a first-class construct and simplifies the implementation of services that need access to this context.

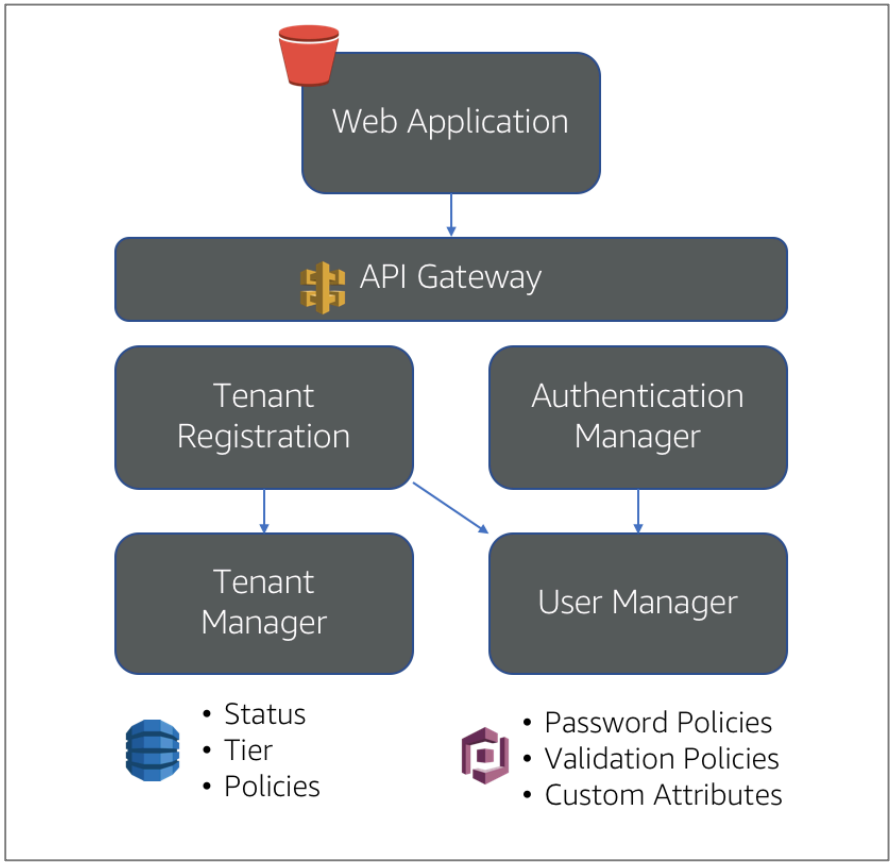
We'll first look at how users get introduced and represented in an identity provider. More specifically, we'll look at how Amazon Cognito can be configured to support the authentication and onboarding flows of our SaaS solution. We'll also use this opportunity to configure user attributes that let us create a connection between users and tenants. This will allow the Cognito authentication process to return tokens that include embedded tenant context.

Once we have users setup, we'll turn our attention to how tenants get represented. Tenants have their own profile and data that is configured separately from the users that are associated with that tenant. We'll introduce a microservice that will own the creation and management of these tenant options (tiering, status, policies, etc.).

With user and tenant management in place, we'll turn our attention to orchestrating the onboarding process. We'll introduce a new registration service that will play this role. We'll also deploy an authentication service that will allow us to authenticate users once

the tenant has been onboarded. Finally, we'll deploy the end-user web application and connect this application to the onboarding system.

By the end of Lab 1, all of the elements will be in place to onboard and authenticate tenants and their users. The diagram below highlights the key elements that are needed to support this experience.



At the top of the diagram is the web application that provides the user interface for our onboarding and authentication flows. This connects to the microservices via an API Gateway. Below the API Gateway are the various microservices needed to support these flows. Tenant Registration is the orchestrator of the onboarding process, invoking the User Manager (to create users in Amazon Cognito) and the Tenant Manager (to create tenants). The Authentication Manager authenticates users via the User Manager service.

What You'll Be Building

As you progress through Lab 1 you'll be creating, configuring, and deploying the elements described above. The following is a breakdown of the steps that you'll be executing to get your SaaS onboarding and authentication experience off the ground:

- Create and Manage User Identity – For this scenario, we'll be using the User Pool construct from Amazon Cognito to manage users and handle user identity. In this

part of the lab we'll take you through the steps to setup the Amazon Cognito environment and deploy the User Management microservice.

- Create and Manage Tenants – Next, we'll create a DynamoDB database that will be used to store our tenant data. Then, we'll deploy the Tenant Management microservice.
- Enabled Tenant Onboarding – Now that we can represent users and tenants, we'll add the Tenant Registration microservice, and configure the web application to connect to the application to the underlying microservices. We'll then onboard a tenant and verify the system created all the appropriate elements.
- Authenticate Users – Once we've created a new tenant, we will be able to authenticate against the system. We'll deploy our orchestration services that own authentication and complete the onboarding process.

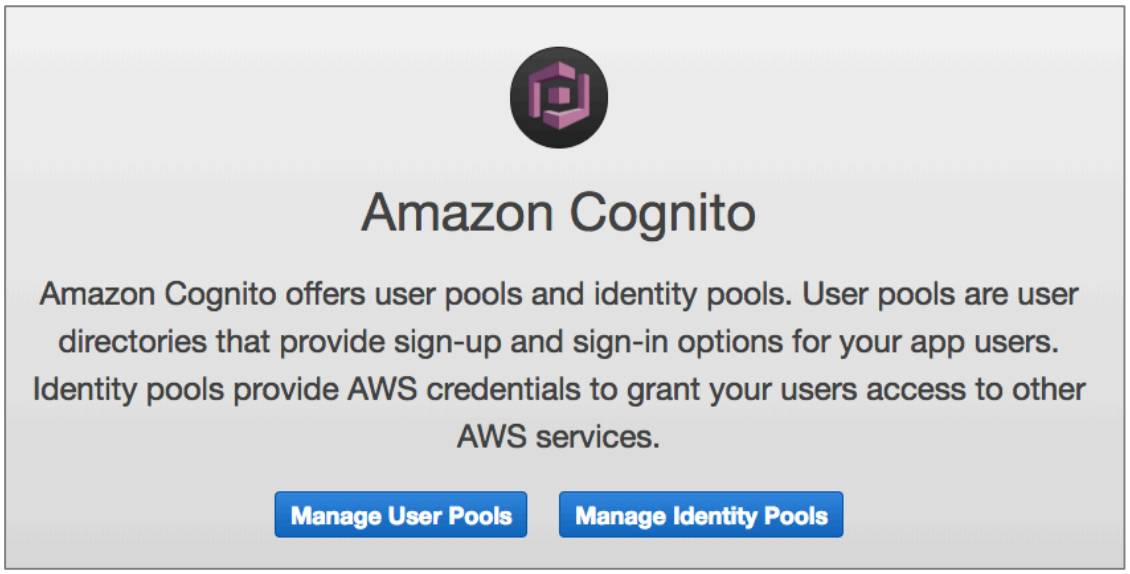
Note: the GitHub repository referenced throughout this document can be found at: https://github.com/aws-samples/aws-saas-factory-bootcamp

Part 1 – Create and Manager User Identity

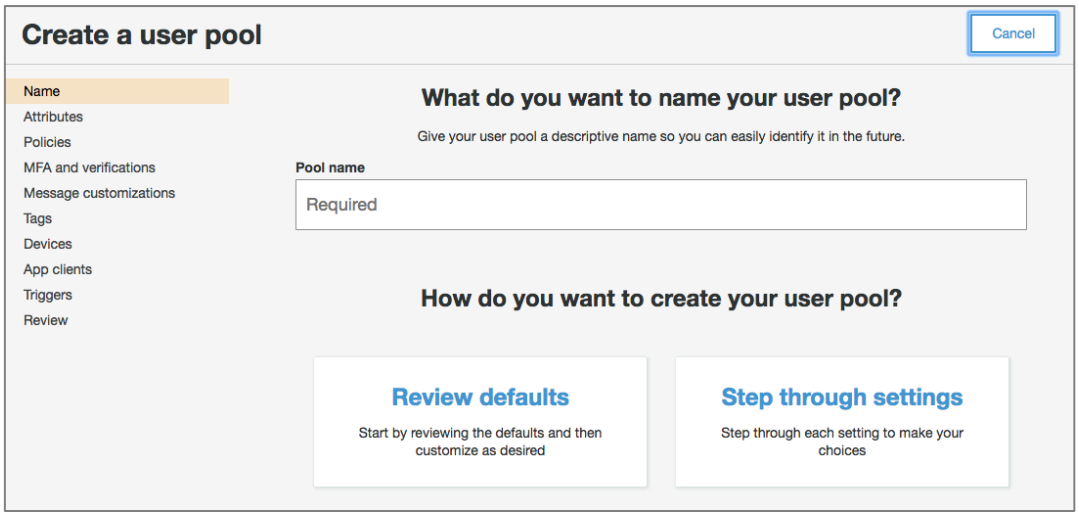
Our goal for this lab is to configure all the elements of Amazon Cognito that will be used to onboard, manage, and authenticate users. We'll also introduce the ability to associate users with tenants that will allow us to create a SaaS Identity. Finally, we'll deploy a User Management microservice to manage our interactions with Cognito.

In this bootcamp, we'll be associating each tenant with a separate user pool. Then, configuring the pool to apply the onboarding and identity settings of our tenants.

Step 1 – Navigate to the Amazon Cognito service in the AWS Console and select "Manage User Pools" from the Cognito Screen.



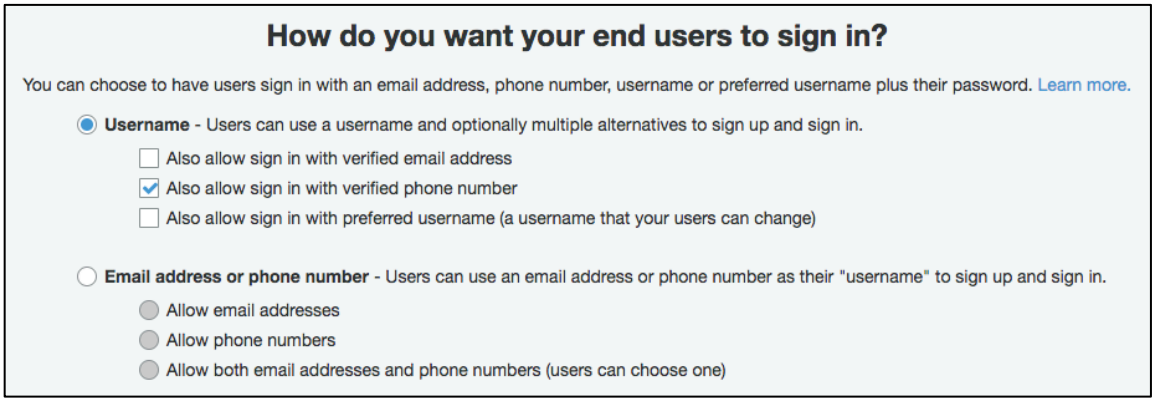
Step 2 – Select "Create A User Pool" from the top right of the Cognito console



Step 3 – You will be presented with a user pool creation experience (shown below) that is used to configure all of the elements of your identity and onboarding experience. The first step in this process is to give your pool a name. Choose any name you'd like (e.g. "SaaS Bootcamp Users").

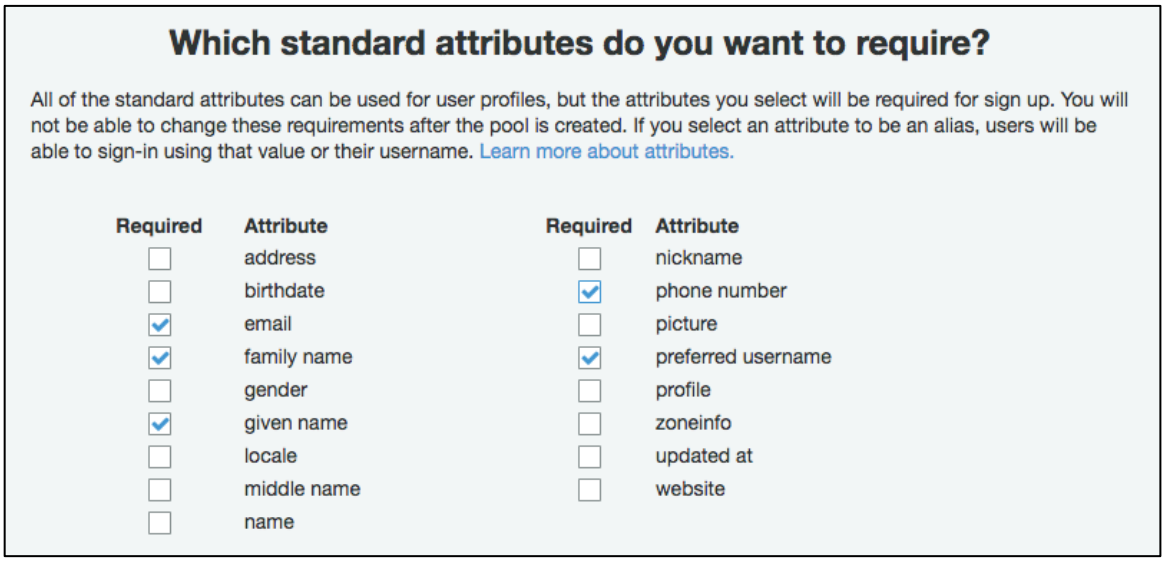
Step 4 – Select "Step through settings" so we can configure the pool to support the onboarding experience we want our SaaS users to have.

The first step allows us to configure the attributes of the pool. Let's start by looking the sign-in policies. The screen below represents the options you'll have.

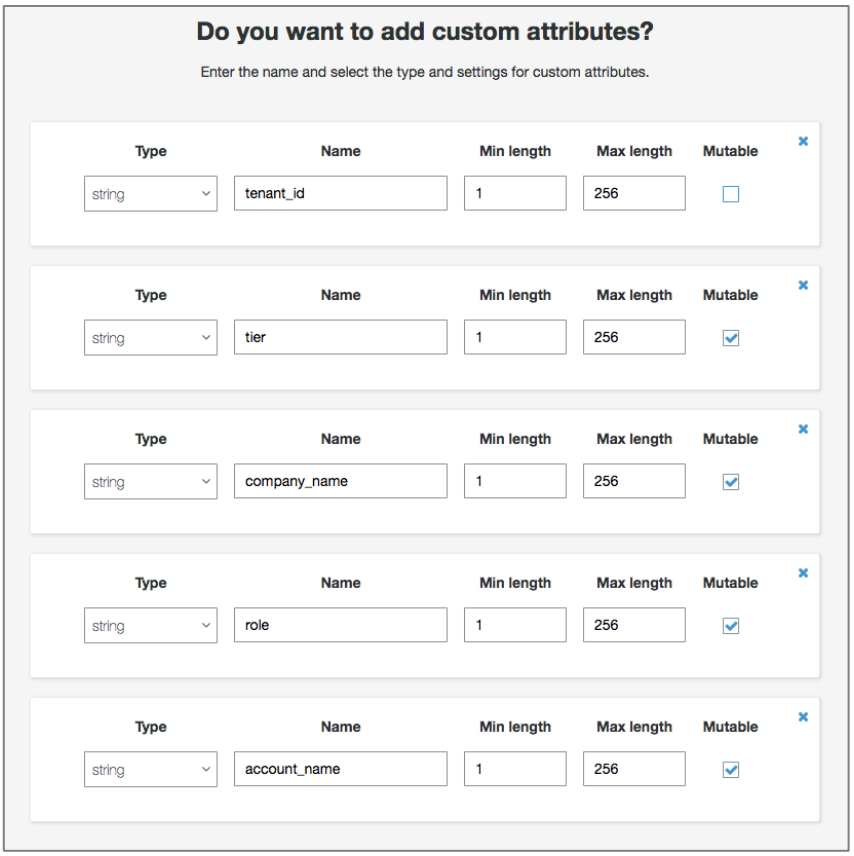


Here we're specifying what options a user can have for their unique identifier (their email in this circumstance). For our solution, we'll check the "Also allow sign in with a verified phone number" option.

Step 5 – Now we move on to the standard attributes portion of the user pool configuration. You are presented with a collection of standardized attributes that are managed by Cognito. Select the email, family name, given name, phone number, and preferred username attributes.



Step 6 – Now we will get more SaaS specific as we turn our attention to configuring the custom attributes of our user pool. This, as stated above, is where we introduce those attributes that will connect users to tenants. When we provision tenants, we'll persist these additional attributes as part of each user's profile. This same



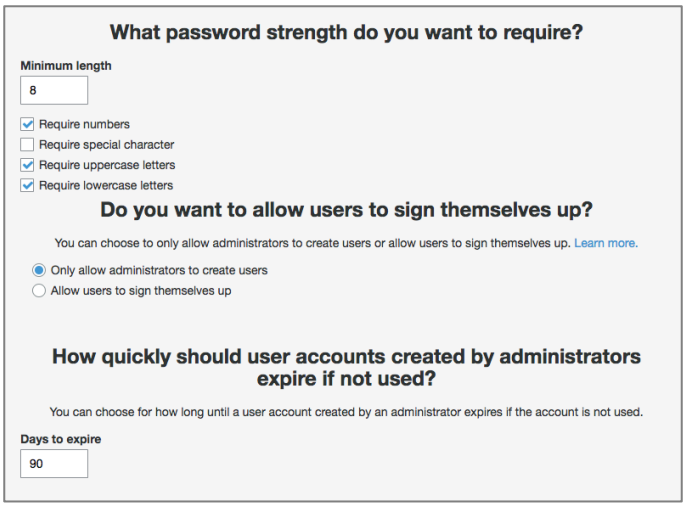
data will also be embedded in the tokens that are returned by the authentication process.

Scroll down the page and click **Add custom attribute**. Add the following tenant attributes we're interested in, selecting **Add another attribute** to for each new attribute to be added:

- **tenant_id** (string, default max length, *not* mutable)
- **tier** (string, default max length, mutable)
- **company_name** (string, default max length, mutable)
- **role** (string, default max length, mutable)
- **account_name** (string, default max length, mutable)

Your screen should appear as follows:

Step 7 – Once you've finished configuring the custom attributes, click the "Next step" button at the bottom of the screen. This takes us to the policies page. Here we can configure password and administration policies. These policies (and others configured with user pools) allow us to vary the approach of each tenant. In fact, these options could, for some solutions, surface in the tenant administration experience of SaaS solutions, allowing individual tenants to configure their own policies.



For our solution, we'll override a few of the default options. First, let's turn off the "Require special character" option for our password policies. Also, let's select the "Only allow administrators to create users" option to limit who can create new users in the system. Finally, set the "Days to expire" for the password to 90 days. The figure above provides a snapshot of the user pool configuration screen with these options configured. Once you've completed this section, click the "Next step" button at the bottom of the page.

Step 8 – We're now at the MFA and verifications section. For Some SaaS providers or providers, or even individual tenants, it could be valuable to enable MFA. For this solution, though, we'll leave it disabled. This page also gives us the option to configure how verifications will be delivered. For this lab we'll leave the default settings. If you choose to enable phone number verification or MFA, Cognito would need an IAM role for permissions to send an SMS message to the user via Amazon SNS.

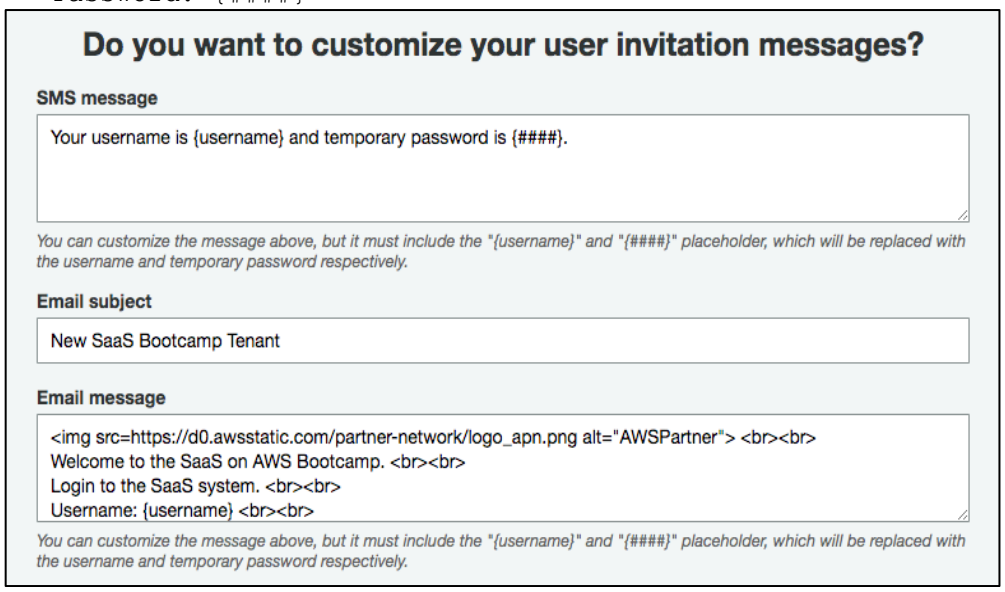
For this lab, just click the "Next step" button.

Step 9 – The 4th step in the wizard is the Message customizations page. As part of our onboarding process, we'll be sending emails to users to verify their identity. We can lean on Cognito for this functionality as well. This screen lets us configure how this verification process will work. For this bootcamp, we will use the default verification message settings. Scroll down the page to the "Do you

want to customize your user invitation messages?" section. Customize the invitation email that will be sent by Cognito as each new tenant signs up as follows:

Change the subject from Your temporary password to "New SaaS Bootcamp Tenant" and the message text to:

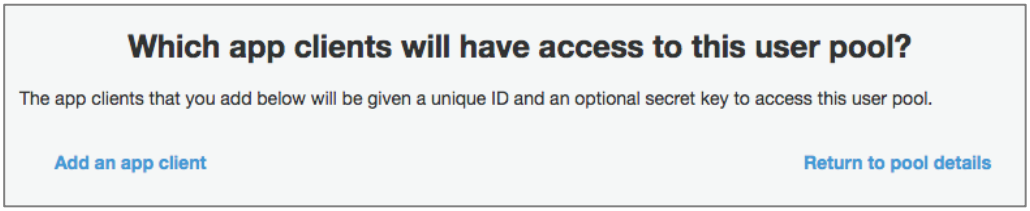
```
<img src=https://d0.awsstatic.com/partner-network/logo_apn.png
alt="AWSPartner"> <br><br>
Welcome to the SaaS on AWS Bootcamp. <br><br>
Login to the SaaS system. <br><br>
Username: {username} <br><br>
Password: {####}
```



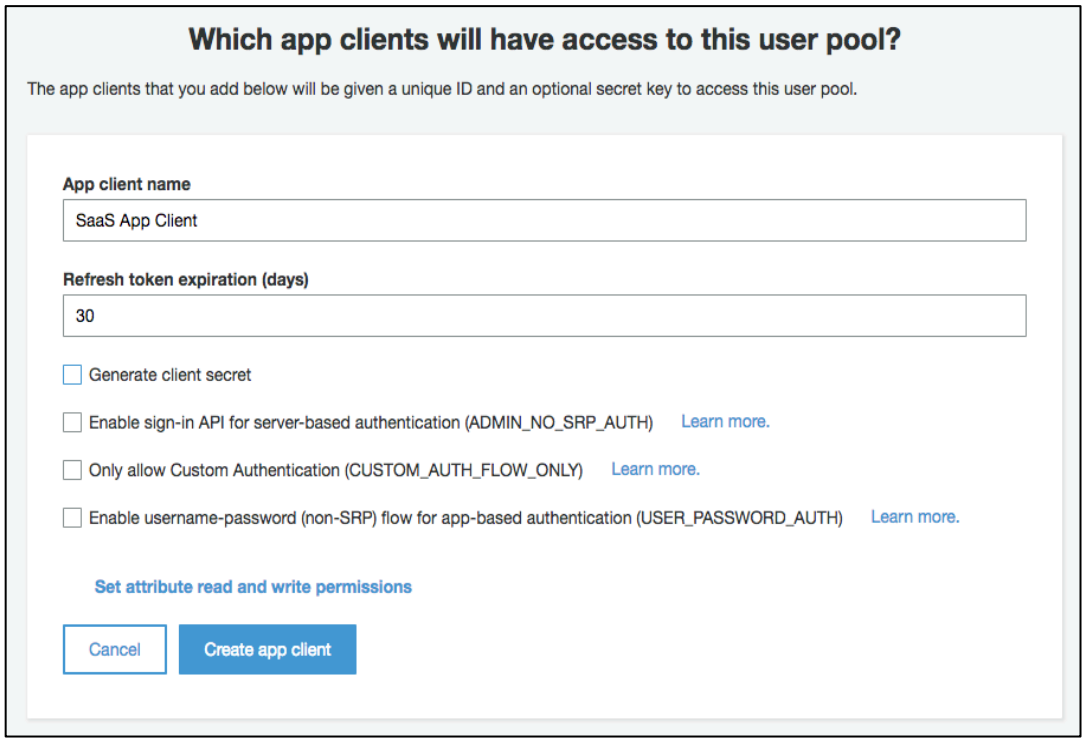
Cognito also has the ability to customize some of the email headers for your verification and invitation emails. We'll leave these settings alone for this bootcamp. Click on the "Next step" button.

Step 10 – For this bootcamp we will skip over the Tags *and* Devices sections. Just click the "Next step" button twice to advance to the App clients screen.

Step 11 – Now that we have the fundamentals of our user pool created, we need to create an application client for this pool. This client is a fundamental piece of Cognito. It provides the context through which we can access the unauthenticated flows that are required to register and sign in to the system. You can imagine how this is key to our onboarding experience. Select the "Add an app client" link from the following screen:

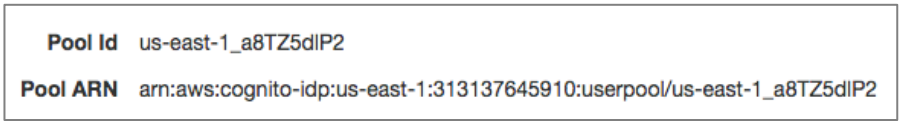


Step 12 – Now we can configure the new application client. Enter the name of "SaaS App Client" and uncheck the "Generate client secret" box. While we're disabling the client secret to simplify this experience, you'd want to enable this option for a production environment. Once you've made these changes, select the "Create app client" button and then the "Next step" button to continue the wizard.



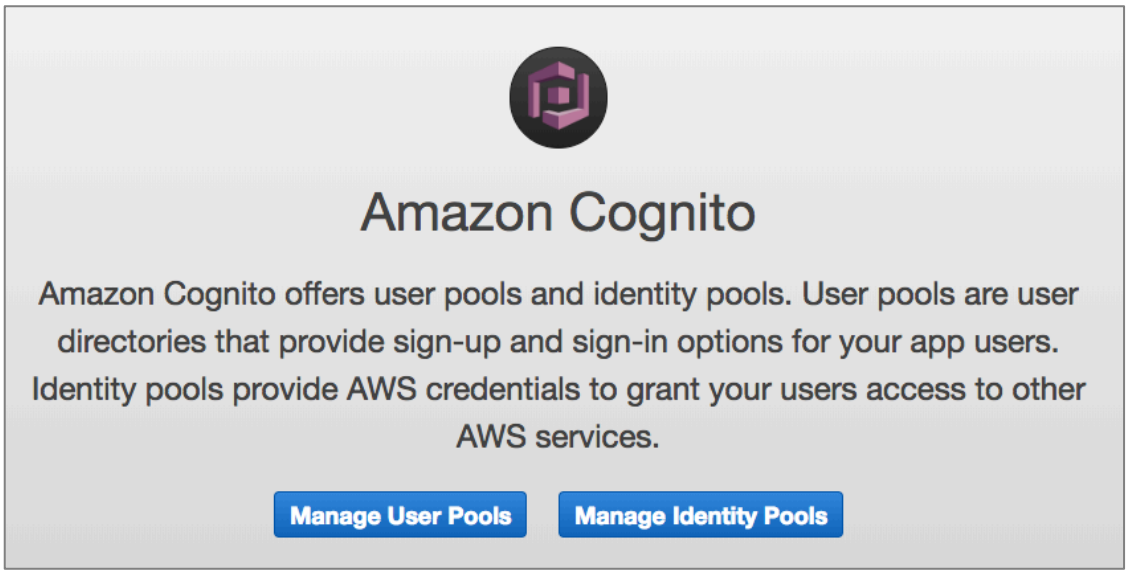
Step 13 – For this bootcamp we will skip over the Triggers section. Scroll to the bottom of the screen and click the "Next step" button to advance to the final review screen and click "Create pool".

Step 14 – Before moving on, we'll want to record both the id that was generated for this user pool and the App client id. Copy and paste the Pool Id value from the General Settings screen into a temporary file or open the next step in a separate web browser window. Also, select the App clients tab from the left-



hand list and save your App client id. We will use both of these values in a subsequent step.

Step 15 – The user pool portion is complete. Before we can use this user pool we'll need to connect it with an identity pool. Identity pools represent the mechanism

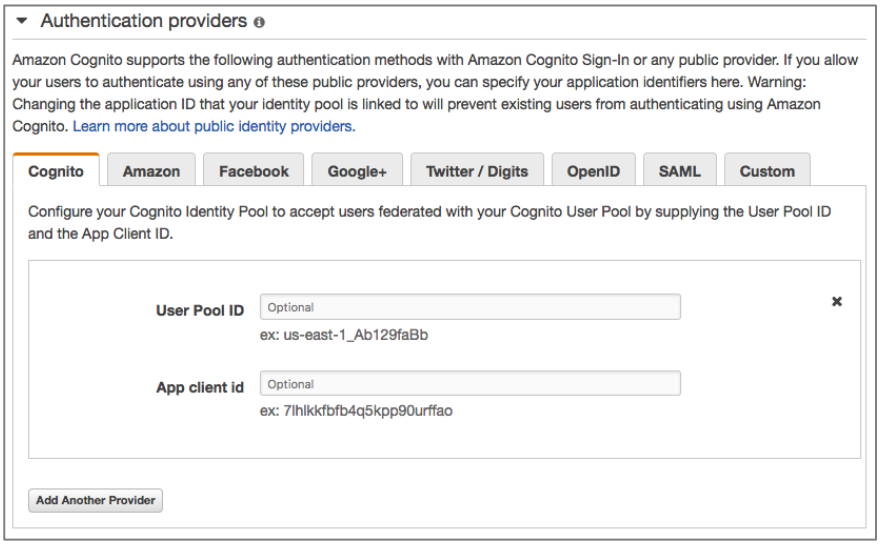


that is used to federate identities from many identity providers and manages our access to AWS resources. To setup your identity pool, navigate back to the main page of Cognito by selecting the AWS icon in the upper left and then selecting Cognito again from the list of. Once here, select the "Manage Identity Pools" button.

Step 16 – The Getting started wizard should launch for you to create a new identity pool because you don't have any existing pools to list.

Step 17 – Enter the name of your new identity pool (e.g. SaaS Identity Pool).

Step 18 – Expand the "Authentication Providers" section at the bottom of the screen by clicking on the triangle. Here's where we'll create the connection between our user pool and the identity pool. You'll see a collection of tabs here representing the various identity providers that Cognito supports. We'll be focusing on the first tab, Cognito. You'll see options here to enter the user pool id as well as the application client id that were captured above. If you don't have them, you can get them by accessing the attributes of the user pool you created above.



Step 19 – Select the "Create Pool" button from the bottom right of the page to trigger the creation of the pool.

Step 20 – Finally, select the "Allow" button on the next page to enable your new identity pool to access AWS resources. This will complete the creation process.

Recap: At this point, you have all the moving parts in place for your SaaS system to manage users and associate those users with tenants. We've also setup the policies that will control how the system validates users during onboarding. This includes the definition of password and username policies. That last bit was to setup an identity pool to enable authentication and access to AWS resources.

Part 2 – Deploying the User Management Microservice

While we've created the AWS infrastructure to support the management of our user identity, we still need some mechanism that allows our application to access and configure these elements of the system. To get there, we need to introduce a microservice that will sit in front of these concepts. This both encapsulates our user management capabilities and simplifies the developer experience, hiding away the details of the Cognito API.

Instead of building this microservice from scratch, we're going to simply deploy this service to ECS. **Note**: the GitHub repository referenced throughout this document can be found at: https://github.com/aws-samples/aws-saas-factory-bootcamp

Step 1 – Before we deploy this service, let's crack open the code and take a closer look at what's here. Navigate to the **Lab1/Part2/app/source/user-manager/src** folder in the GitHub repo for this bootcamp and examine the **server.js** file. This file is a Node.js file that uses the Express framework to implement a REST API for managing users. Below is a list of some of the entry points that may be of interest for this onboarding flow.

```
app.get('/user/pool/:id', function (req, res)
app.get('/user/:id', function (req, res)
app.get('/users', function (req, res)
app.post('/user/reg', function (req, res)
app.post('/user/create', function (req, res)
app.post('/user', function (req, res)
app.put('/user', function (req, res)
app.delete('/user/:id', function (req, res)
```

These represent a handful of the entry points into the user manager service. It includes basic CRUD operations in addition to functionality to support registration and fetch of user pools.

Step 2 – Since Cognito will serve as the repository to store our users, the user manager service must make calls to the Cognito API to persist new users that are created in the system. To illustrate this, let's take a closer look at an initial version of the POST method in user manager that will persist users to Cognito.

```
316  app.post('/user/create', function (req, res) {
317      var newUser = req.body;
318
319      var credentials = {};
320      tokenManager.getSystemCredentials(function (systemCredentials) {
321          if (systemCredentials) {
322              credentials = systemCredentials;
323
324              cognitoUsers.createUser(credentials, newUser, function(err, cognitoUsers) {
325                  if (err) {
326                      res.status(400).send('{"Error": "Error creating new user"}');
327                  } else {
328                      res.status(200).send(cognitoUsers);
329                  }
330              });
331
332          } else {
333              res.status(400).send('{"Error": "Could not retrieve system credentials"}');
334          }
335      });
336  });
```

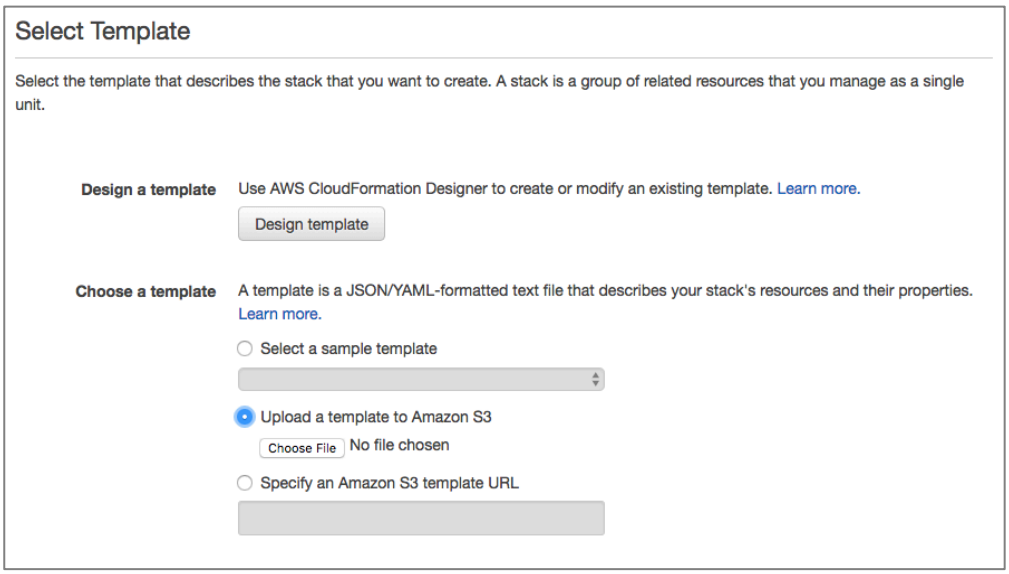
Here you'll see that our POST method accepts a user JSON object that is then passed along to Cognito to create the user in our user pool.

Step 3 – Now that have a clearer view of what's happening behind the scenes, let's deploy this user manager microservice. We have prepared a **CloudFormation** template to build the Docker image of our microservice and deploy it to **Elastic Container Service**.

Download the **user-service.template** file to your computer from the GitHub repository for this bootcamp at Lab1/Part2/templates/user-service.template. You can right-click on the "**Raw**" button in GitHub and select "**Save Link As**".

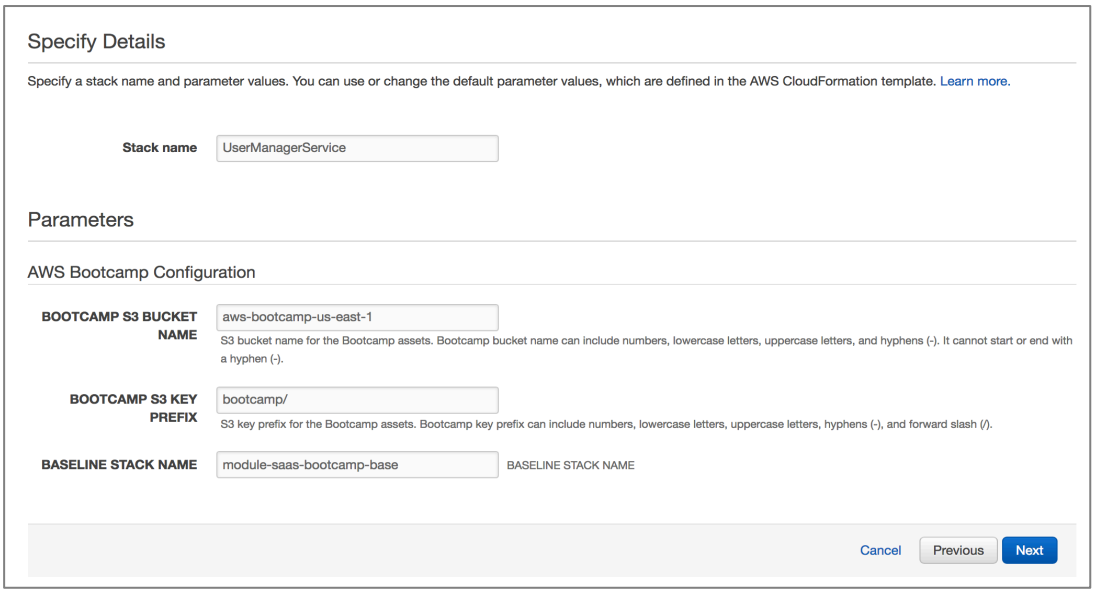
Step 4 – Navigate to the CloudFormation service in the AWS console and select the "**Create Stack**" button at the upper left of the page.

Step 5 – After you have selected create stack you'll be presented with a screen of option for selecting a template to be provisioned. The screen will appear as follows:

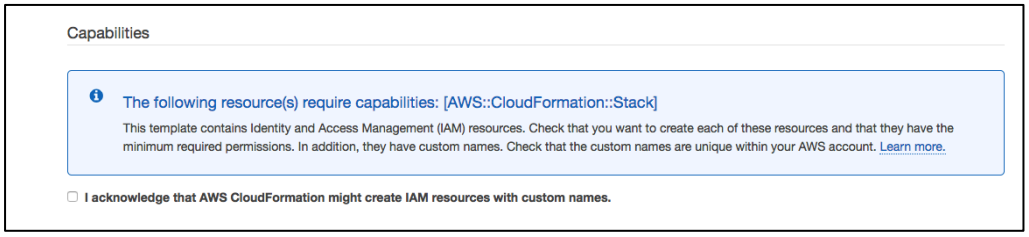


Here you'll see multiple approaches to uploading a template file. Select the "**Upload a template to Amazon S3**" option. Then, select the "Choose File" button and select the template file you downloaded in Step 3. Finally, select the "**Next**" button to move to the next step in the creation process.

Step 6 – The CloudFormation service will now prompt you for a stack name. Enter "**UserManagerService**" for the stack name and leave the other parameters with their default values. For example:



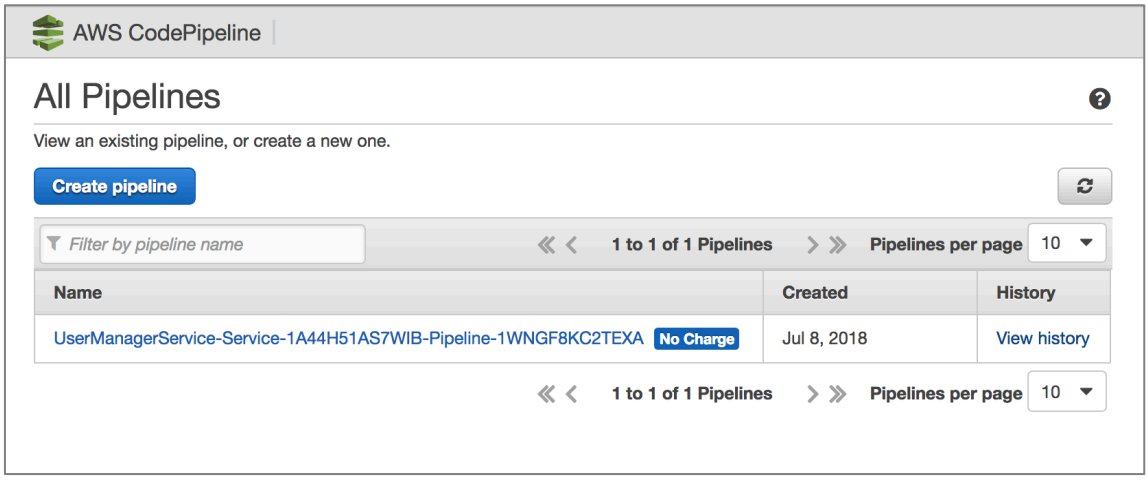
Select the "Next" button to initiate the creation of the user management service and select "Next" once more on the options page. Finally, review the summary. At the bottom of the page, select the "I acknowledge that AWS CloudFormation might create IAM resources with custom names" checkbox.



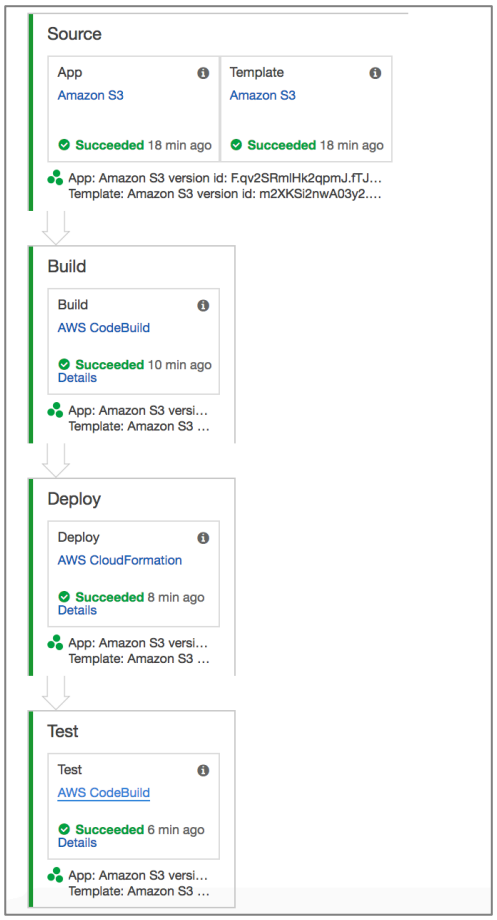
The last step is to select the "Create" button at the bottom right of the summary page to trigger the execution of the stack. You'll see a number of Stacks start to execute and be in the CREATE_IN_PROGRESS status. Highlighting individual rows in the stacks table will allow you to view the events being triggered by that stack.

Step 7 – After a couple of minutes, navigate to CodePipeline in the AWS console and you'll see the microservice being built and deployed to ECS.

> Note: the pipeline may not display immediately. Continue to refresh the page and, eventually, the CloudFormation will get far enough along in its process that the pipeline will be visible in the console.



Step 8 – In CodePipeline click on the pipeline labeled with a prefix of UserManagerService.

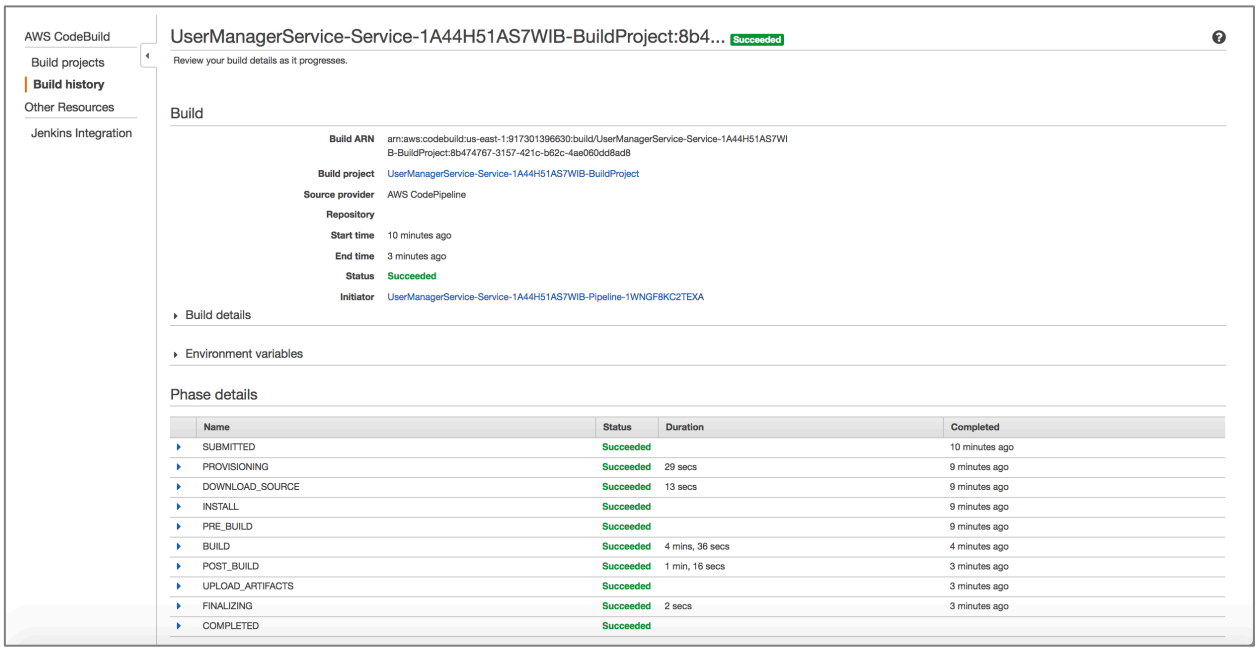


Step 9 – Notice, how the pipeline consists of the following sequences:

Source Sources Code from an S3 Bucket in AWS Account

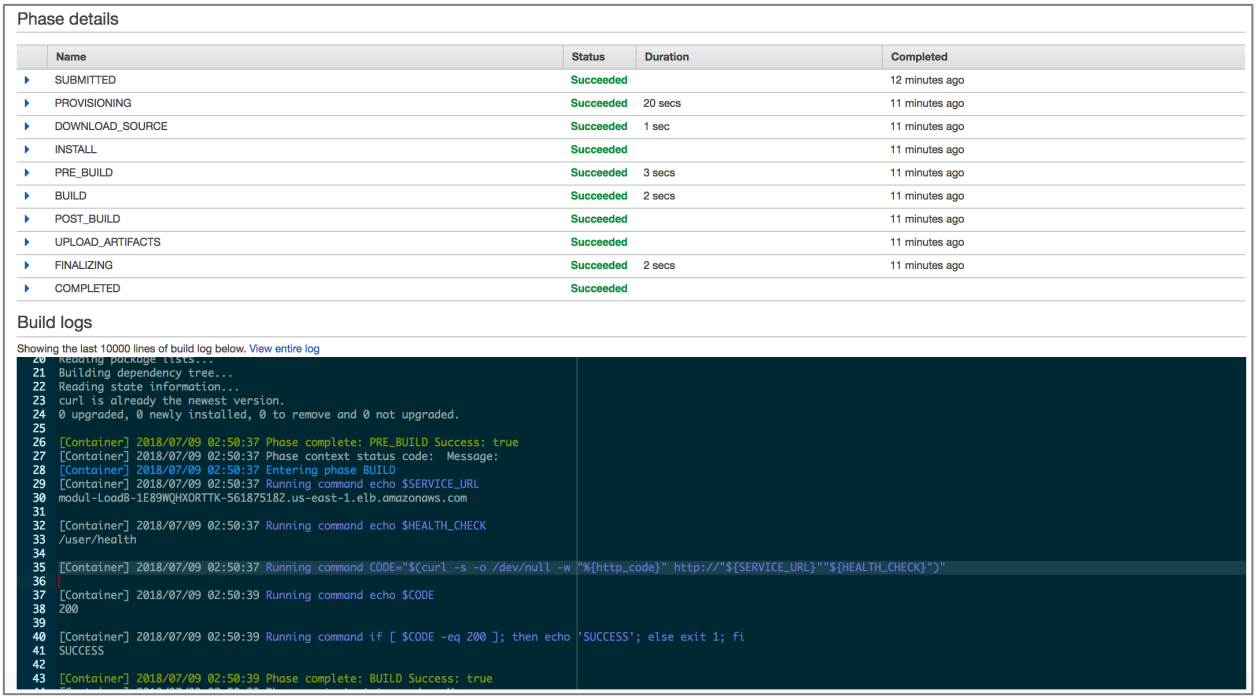
Build	Build and Push Docker Container to Elastic Container Registry
Deploy	Deploys microservice to an ECS Service via CloudFormation
Test	Completes a basic http health check to verify service upti

Step 10 – Shortly after **CodePipeline** is invoked and the source code is downloaded from the originating S3 Bucket, a **CodeBuild** Project is fired off. Click the word "**Details**" under the In Progress animation in the **Build** card to view the details of this build process. Scroll down and you'll see the build log being updated in real time as the process completes.

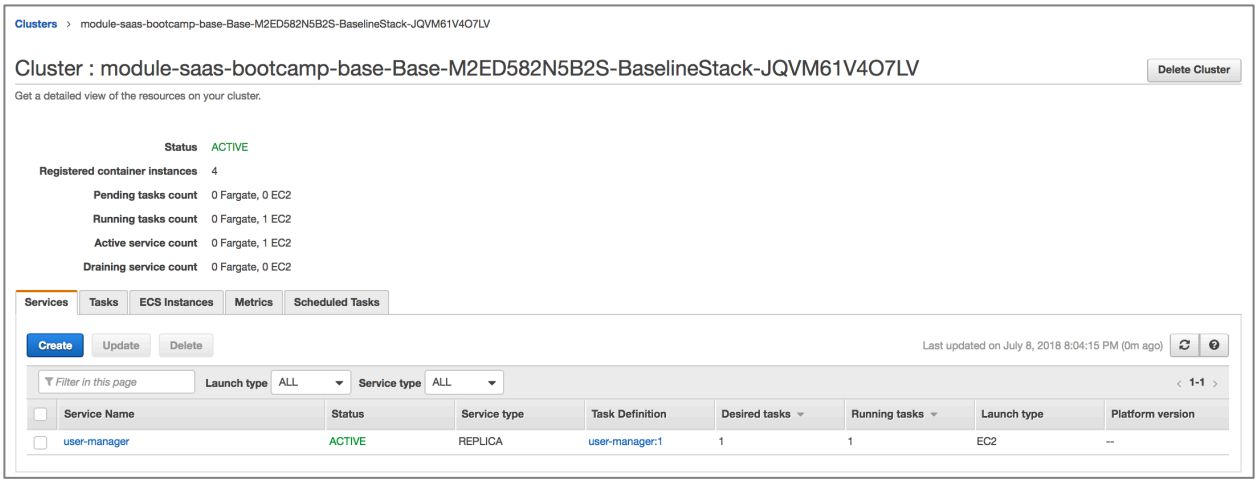


Step 11 – Once the **CodeBuild** Project has completed we have a deployment step that will create the corresponding ECS Service for our **UserManagerService**.

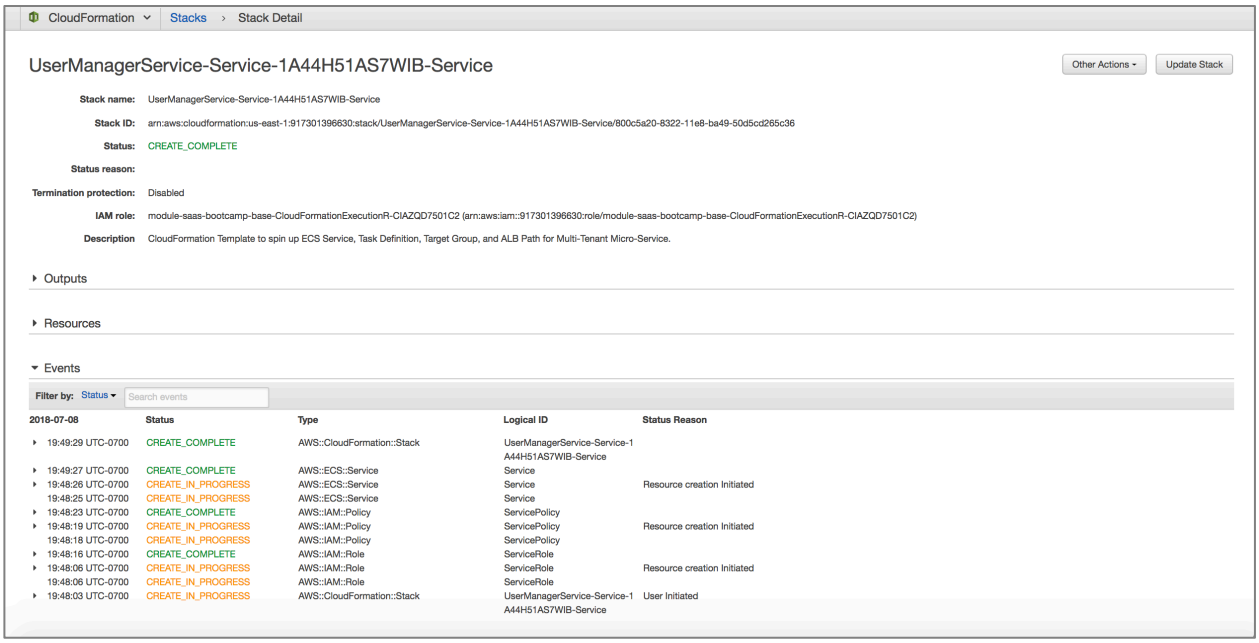
Step 12 – The final step in our Code Pipeline is to execute a verification step that the microservice is indeed healthy within **Elastic Container Service**. For simplicity, the **Test** step is invoking a **GET** method via a **cURL** to the microservice endpoint **/user/health**.



Step 13 – When finished, navigate to the **Elastic Container Service** and note that the user management service has been created and a task launched onto one of the instances in the cluster.

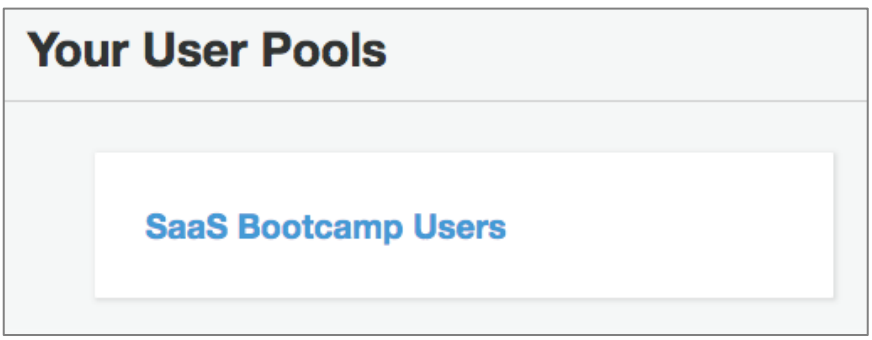


You can also confirm successful completion of the CodePipeline by examining the state of your CloudFormation stack. If you navigate back to the CloudFormation service in the AWS console and locate the **UserManagerService** stack, you'll notice that it has completed (as shown below).



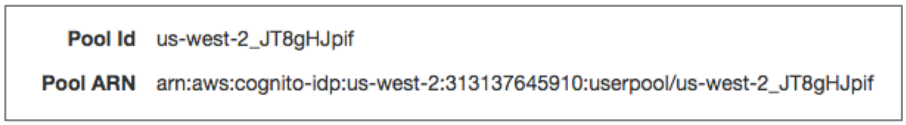
Step 14 – With the user manager service up-and-running, you can now make a call to this REST service to create a user in the user pool we created for our tenant. To achieve this, you'll need the URL of the Application Load Balancer that our ECS cluster is behind and the Pool Id from Cognito that you saved earlier when creating the identity pool. **Navigate to the EC2 console** listed under the Compute heading in the AWS console. Scroll down the left-hand menu and select **Load Balancers**. In the Description tab you will see the public **DNS name** we can use to invoke our microservices. For example:

Step 15 – If you need to retrieve your user pool id again, navigate to the Cognito service in the AWS console and select the "Manage User Pools" button. You will be presented with a list of user pools that have been created in your account (as shown below).



Select the user pool that you created earlier to display information about the user pool.

Step 16 – After you've selected the pool, you'll be presented with a summary page that identifies the attributes of your user pool. The data you're after is the Pool Id, which is shown at the top of the page (similar to what is shown below).



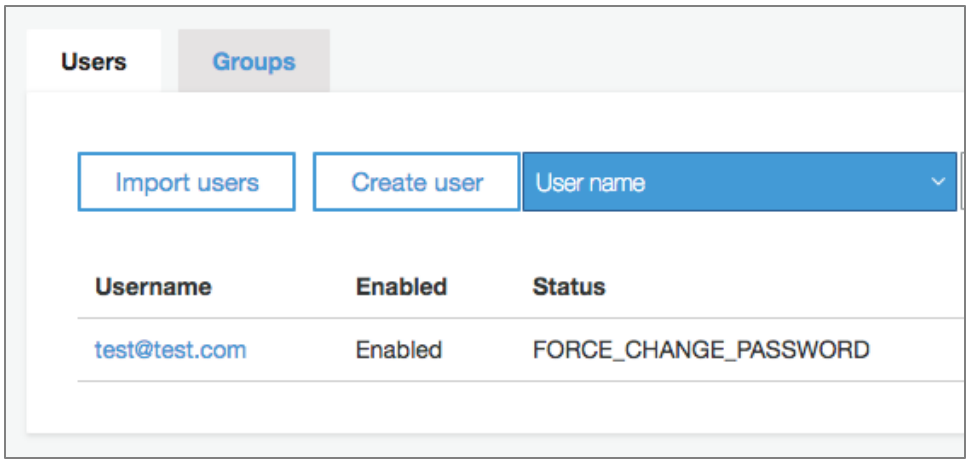
Step 17 – Now that you have the pool id and the load balancer URL you're ready to call the REST method on the user manager service to create a user in Cognito. To call our REST entry point, we'll need to invoke the create user POST method. You can do this via a variety of tools (cURL, Postman, etc.). Below we've shown how to issue this command from the terminal with cURL. Be sure to enter in your userPoolId and a valid email address for the username and email attributes.

```
curl --header "Content-Type: application/json" --request POST --data
'{"userPoolId": "USER-POOL-ID", "tenant_id": "999", "userName":
"test@test.com", "email": "test@test.com", "firstName": "Test",
"lastName": "User", "role": "tenantAdmin", "tier": "Advanced"}'
http://LOAD-BALANCER-DNS-NAME/user/create
```

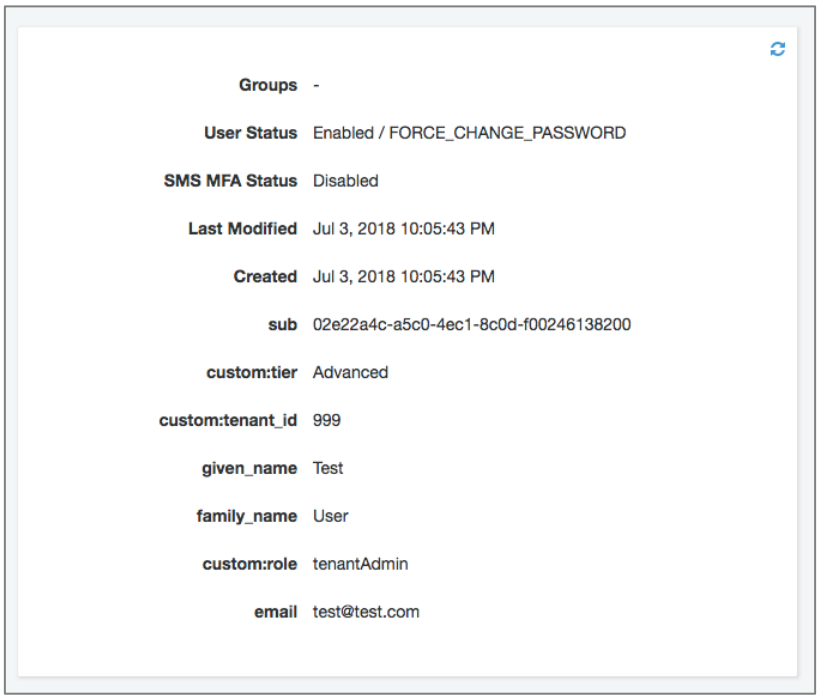
On success, the newly created user object will be returned in JSON format.

Step 18 – You can now verify that your new user was created in the Cognito user pool. Once again, let's return to the Cognito service in the AWS console. After selecting the service, select "Manager User Pools" and select the user pool created above to drill into that pool.

Step 19 – Select "Users and groups" from the menu on the left. When you select this option, you'll see the list of users in your pool (as shown below).



Step 20 – Your newly added user should appear in this list. Select the link for your username to get more detailed information about the user. A screen similar to the following will be shown.



Here you'll be able to see how you user landed in the system with the tenant, name, and email address you provided via your REST command.

Step 21 – If you used a valid email address you have access to, you should also have received the Invitation Email you configured for the user pool in Cognito at the beginning of this lab.

Recap: This phase was relatively straightforward. The key goal here was to demonstrate how microservices were being introduced and how the user management service,

specifically, was abstracting away the details of the Cognito API. You also were able to see how the user management service was able to create new users in your user pool, and how build and deployment takes place through CodePipeline, CodeBuild, and CloudFormation. While we've focused here on performing user pool and user creation as manual step, the final version of this solution will automate the creation of the user pool for each tenant during onboarding.

Part 3 – Managing Tenants

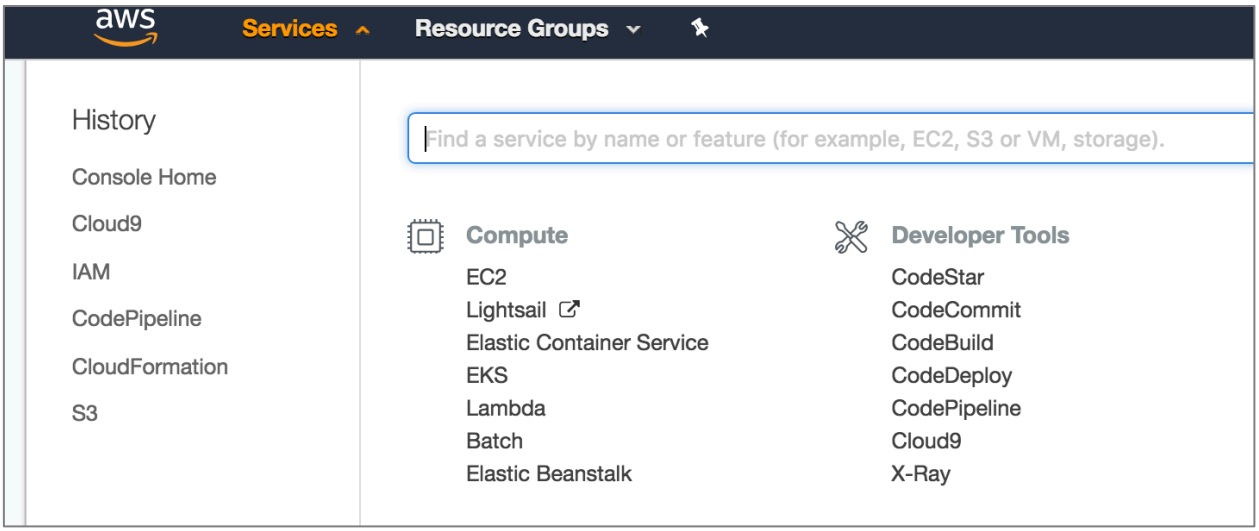
At this point, we have a way to create users as part of the onboarding process. We also have a way to associate these users with tenants. What we're missing is some ability to store and represent tenants. Tenants must be represented and managed separate from users. They have policies, tiers, status, and so on—all of which should be managed through a separate contract and service.

Fortunately, the management of these services is relative straightforward. It simply requires a CRUD microservice that will manage data stored in a DynamoDB table. The following steps will get our tenant manager microservice deployed and ready for consumption.

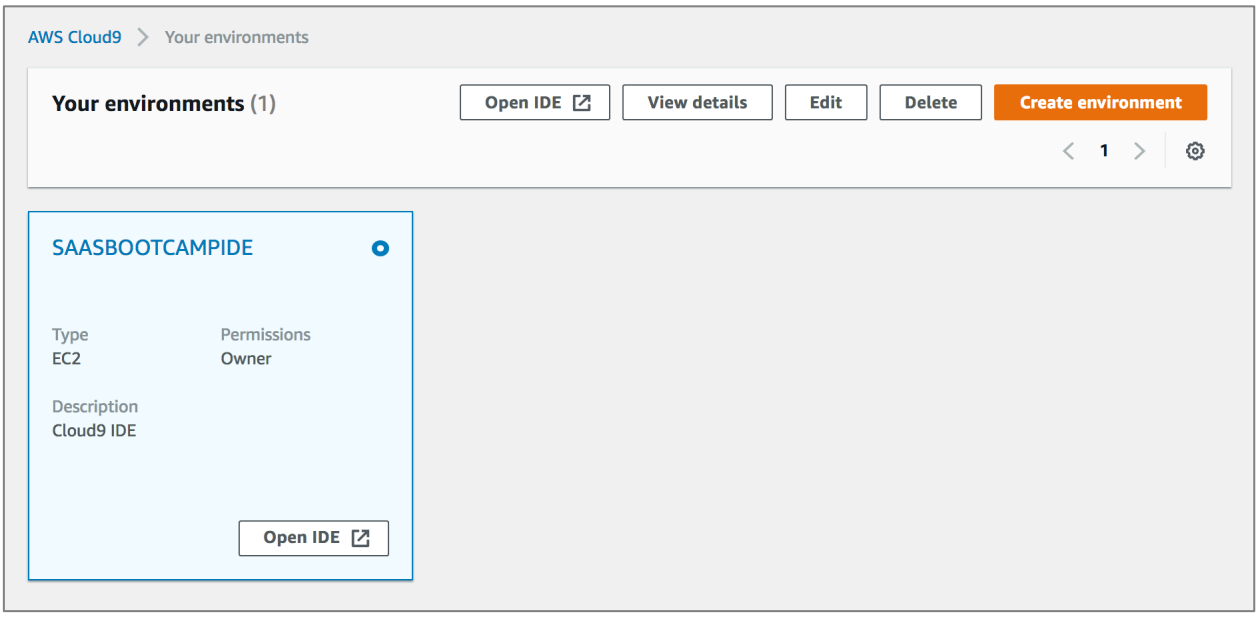
Step 1 – For much of the rest of the bootcamp, we will be leveraging AWS Cloud9, an integrated development environment, so we can make changes more rapidly, and through a local development environment prior to deploying into AWS.

Step 2 – Our first step is to deploy the Tenant Manager Service to a container running as part of an ECS cluster. We will run a local shell script to build the microservice. This script uses Docker on our Cloud9 development workstation to build the container from the code sourced within our GitHub repository. The script then uses AWS command line tools to deploy the microservice to our ECS cluster.

Step 3 – To get started, login to the AWS Cloud9 IDE within the AWS Console. From the AWS Console home page click the "Services" drop-down and click on Cloud9.

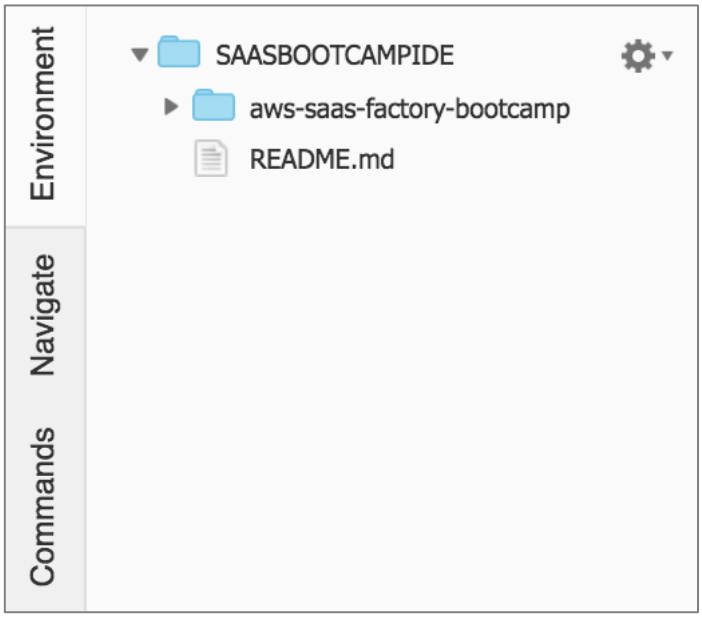


Step 4 – Next, you should see a screen labeled "Your environments" with an environment labeled SAASBOOTCAMPIDE.

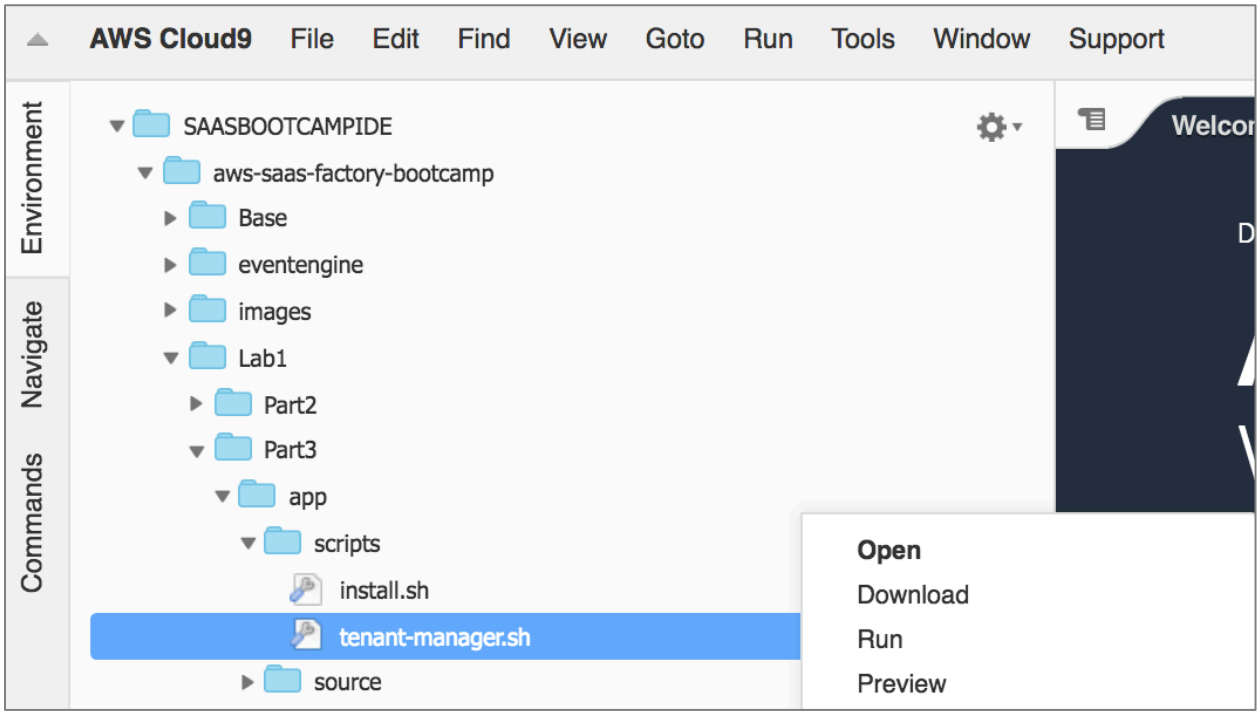


Step 5 – Click the button "Open IDE", which should open a new browser tab. Note, this tab will take a couple moments to load and start your Integrated Development Environment (IDE). As part of this launch, the IDE will automatically clone the GitHub repository for this bootcamp.

Step 6 – Once your Cloud9 IDE environment has fully loaded, navigate to the left-hand side where a folder named **aws-saas-factory-bootcamp** is present. This folder contains all the contents from the GitHub repository we have been leveraging up till now.



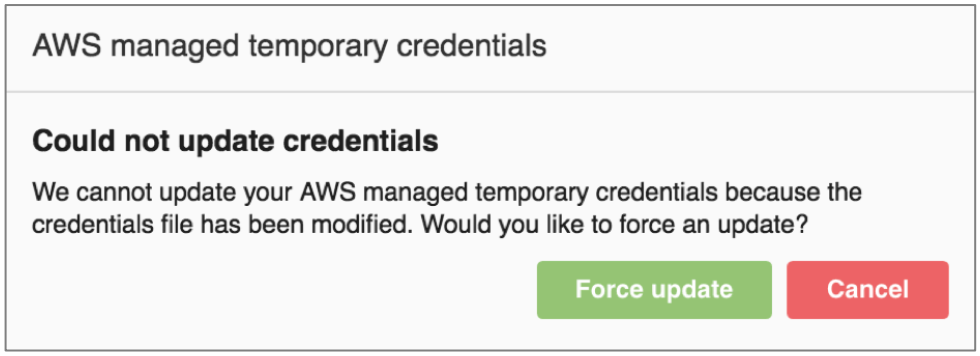
Step 7 – Open the tree structure for the **aws-saas-factory-bootcamp** folder, and navigate into the folder **Lab1/Part3/scripts/** and right-click on the file **tenant-manager.sh** and click **Run**.

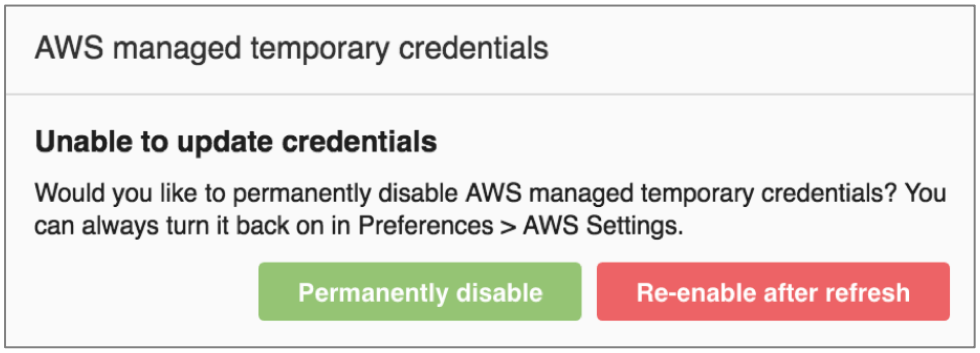


Step 8 – This **tenant-manager.sh** script will build our tenant manager microservice code in a Docker container image and subsequently push this container image into **Elastic Container Registry (ECR).** Note, in addition to building and pushing the

container image to **ECR**, this setup script will be responsible for initializing the setup of your **Cloud9** IDE environment. The first time this script runs it will run longer than the subsequent shell scripts in this Lab. This initial longer process time is a result of some initial housekeeping, and the Docker process taking an iterative approach to building containers through intermediary images. We will take advantage of this iterative design when we rebuild existing container images as we progress through the labs in this bootcamp.

Note, as we are overriding the credentials you may get an error message stating "**Could not update credentials**". If so, please click the **Cancel** button, followed by "**Permanently disable**".





Step 9 – Let the **tenant-manager.sh** script run until it has successfully completed as noted by the line "Process exited with code: 0", with a message "**STACK CREATE COMPLETE**". This will take a few minutes. If the script fails, please contact a bootcamp facilitator to troubleshoot.

Step 10 – With the tenant manager service up-and-running, you can now make a call to this REST service to create a new tenant. To achieve this, you'll need the URL of the Application Load Balancer that our ECS cluster is behind. Navigate to the EC2 console listed under the Compute heading in the AWS console. Scroll down the left-hand menu and select Load Balancers. In the Description tab you will see the public DNS name we can use to invoke our microservices. For example:

Step 11 – Now let's exercise the new tenant management service to confirm that it's working. We'll use a cURL command (or Postman, if you prefer) to create a new tenant in the system.

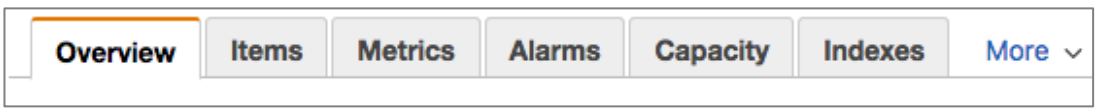
```
curl --header "Content-Type: application/json" --request POST --data
'{"id": "111", "role": "tenantAdmin", "company_name": "Test SaaS
Tenant", "tier": "Advanced", "status": "Active"}' http://LOAD-
BALANCER-DNS-NAME/tenant
```

Step 12 – This command will have created an item in the DynamoDB table that holds our tenants. Let's navigate to the DynamoDB service within the AWS console and verify that the tenant was created.

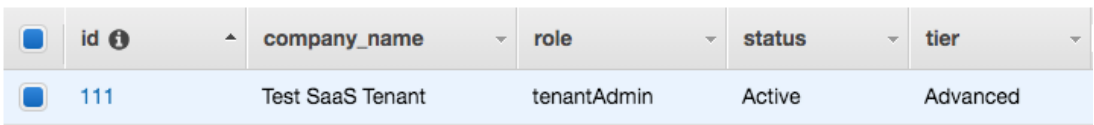
Step 13 – Once you're in the DynamoDB console view, select "Tables" from the menu on the upper left-hand side of the page. This will get you access to all the tables in your DynamoDB account.

Step 14 – Locate and select the "TenantBootcamp" table hyperlink from the list of DynamoDB tables.

Step 15 – You will now have an overview the attributes of the table. Let's select the "Items" tab from the top of the page to see the actual data in the table.



Step 16 – The item you added should now appear in the table. A sample is shown below:



Recap: The goal of this section was merely to introduce you to the tenant manager service and the separate representation of tenant data. The tenant identifier in this DynamoDB table will be associated with one or more users via the tenant_id custom attribute that we created in the prior section. By separating the unique tenant data out from our user attributes, we have a clear path for how tenants are managed and configured.

Part 4 – Deploying the Tenant Registration and Authentication Services

User and tenant management is now resolved. Still, we need a service that can orchestrate the entire onboarding flow spanning the user and tenant creation process. This is achieved through the tenant registration and authentication services. These two services play different roles in the onboarding process. The registration service handles the initial provisioning and the authentication service is used to authenticate a new tenant's temporary credentials. The authentication service is also used for subsequent attempts to authenticate users after provisioning is completed.

For the purposes of this lab, we're simply going to deploy these services and connect to the onboarding application (via the Application Load Balancer).

Step 1 – Before we deploy this service, let's crack open the code and take a closer look at what's here. Using the Cloud9 IDE, navigate to the Lab1/Part4/app/source/tenant-registration/src folder and examine the server.js file. This file is a Node.js file that uses the Express framework to implement a REST API for managing the registration process. They key entry point in this service is the registration method which orchestrates the business logic flow and delegates to the User and Tenant management services that were deployed earlier.

```
app.post('/reg', function (req, res)
```

Here's a snippet from the code invoked by the HTTP Post:

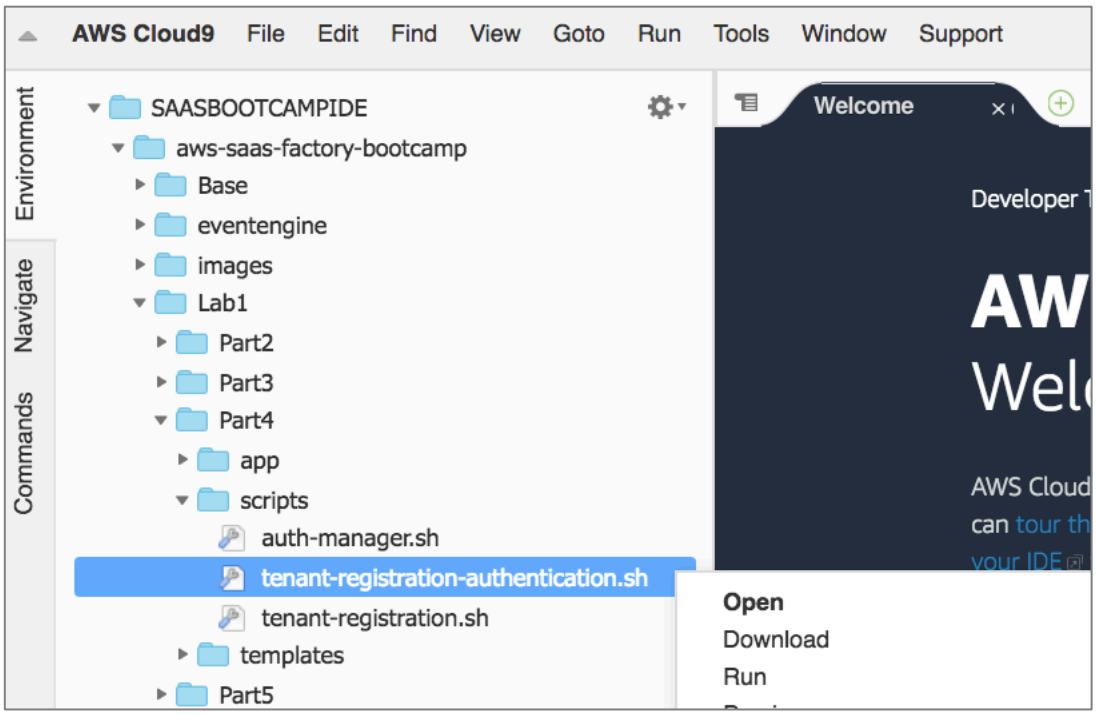
```
// First create the tenant admin user via the
// User Manager Service
registerTenantAdmin(tenant)
    .then(function (tenData) {
        // Adding Data to the Tenant Object that will be required for
        // cleaning up all created resources for all tenants.
        tenant.trustRole = tenData.role.trustRole;
        tenant.systemAdminRole = tenData.role.systemAdminRole;
        tenant.systemSupportRole = tenData.role.systemSupportRole;
        tenant.systemAdminPolicy = tenData.policy.systemAdminPolicy;
        tenant.systemSupportPolicy = tenData.policy.systemSupportPolicy;

        // Keep track of where all the users are for this tenant
        tenant.UserPoolId = tenData.pool.UserPool.Id;
        tenant.IdentityPoolId = tenData.identityPool.IdentityPoolId;

        // Now save the tenant attributes to our data store
        // via the Tenant Manager Service
        saveTenantData(tenant)
```

Step 2 – The entry points for the authentication service are as you might suspect. If you navigate to the **Lab1/Part4/app/source/auth-manager/src** folder and examine the **server.js** file, you'll find the Node.js file that uses the Express framework to implement a REST API for authentication. The main entry point here is the "/auth" method. Here's the signature for that method:

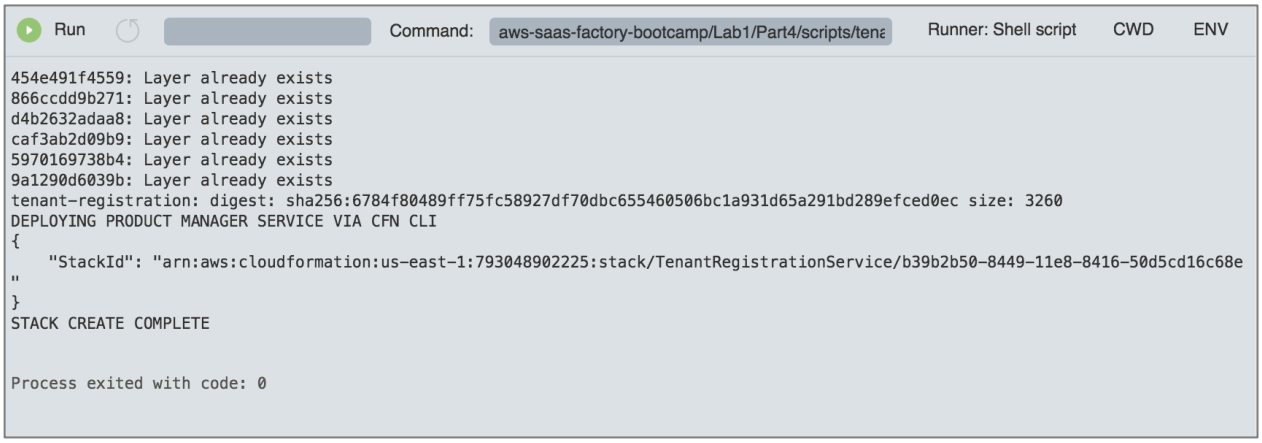
```
app.post('/auth', function (req, res)
```

This POST method on the service accepts the user, authenticates the user, and returns our SaaS identity tokens.

Step 3 – Now it's time to deploy the tenant registration and authentication microservices. The goal here is to deploy these services to containers that are running as part of an ECS cluster. For simplicity's sake, we'll deploy these services using a single shell script. Navigate to the directory **Lab1/Part4/scripts/** and execute the shell script **tenant-registration-authentication.sh** in Cloud9 by right-clicking and selecting **Run**.



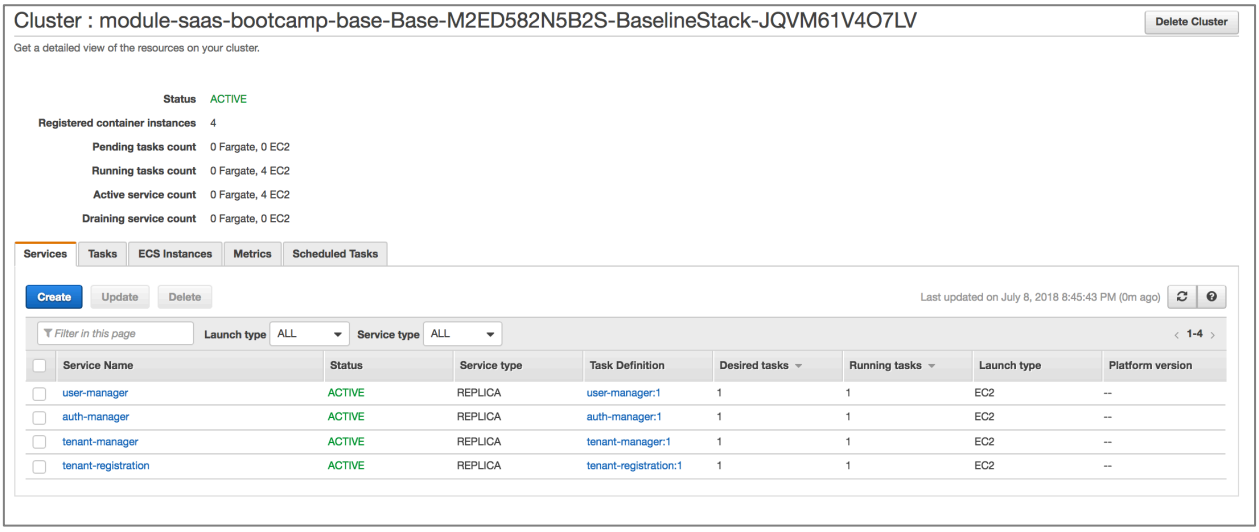
Step 4 – Note, that since this shell script will be creating two micro-services it will take a few minutes to complete, however, notice that two services were successfully built faster than the single service was in the previous step. Ensure that you receive a successful completion by verifying “Process exited with code: 0”.



This speed-up is as a result of Docker leveraging existing container images in its iterative build process.

```
9958cf002363: Layer already exists
3706a3bcd0b1: Layer already exists
e5796e6a8531: Layer already exists
2ff2d7030bda: Layer already exists
fdd2ae61e2e2: Layer already exists
7f630c80ecbf: Layer already exists
e0faf32572e1: Layer already exists
9a1290d6039b: Layer already exists
```

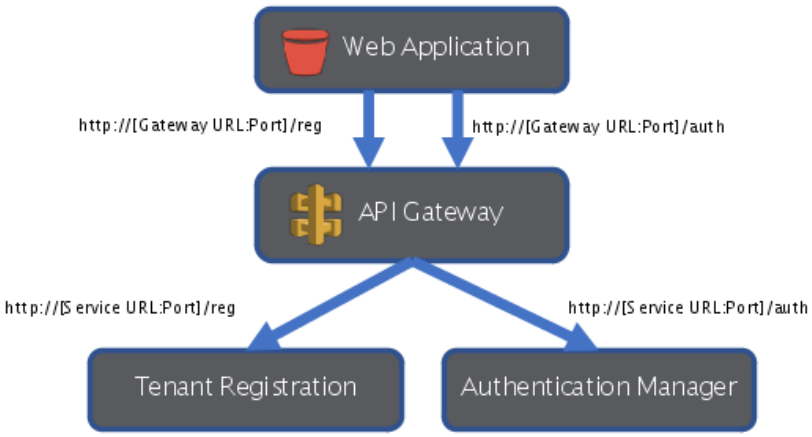
Step 5 – Navigate to the **Elastic Container Service** in the **AWS console** and note that the tenant registration and authentication services have been created.



Recap: This phase was all about getting the registration and authentication services in place and making the registration service available for the onboarding application.

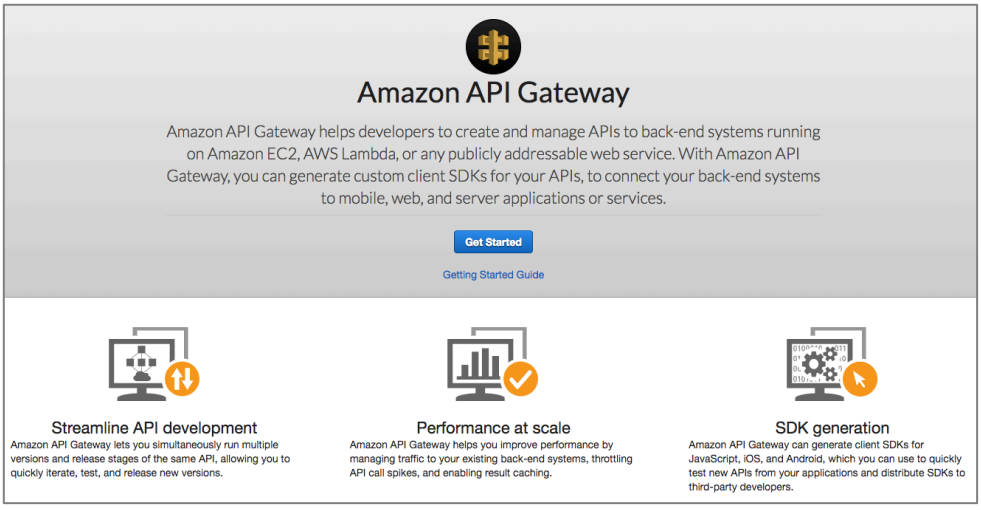
Part 5 – Configure API Gateway for Authentication & Registration

At this stage everything is about ready for us to onboard new tenants. It is now necessary for us to configure the **Methods** and **Resources** for the underlying authentication and registration microservices via the **API Gateway**. This is purely a mapping exercise where we map a URL to the entry points of the corresponding **REST** methods in each of the microservices. The **API Gateway** URLs will be constructed with a model that matches the URLs that are already configured in the application. The diagram below provides a conceptual view of this mapping.

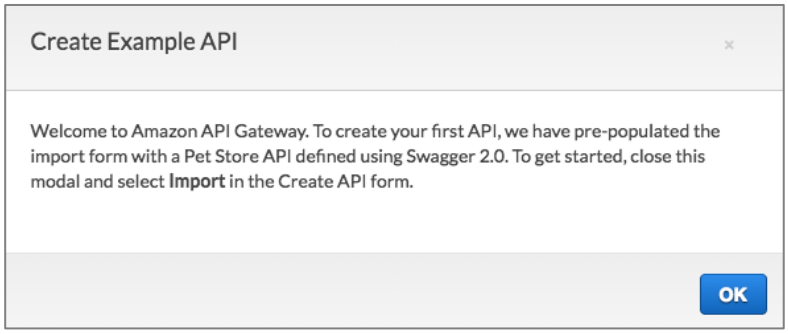


In this diagram we have the URLs that the exposes for our application. These entry points are then mapped to the underlying registration and authentication microservices.

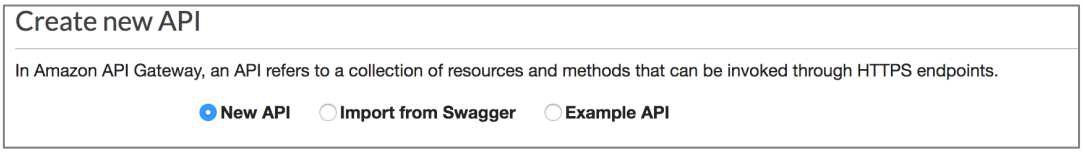
Step 1 – To start this mapping process, we must first navigate to the Amazon API Gateway service in the AWS console. After accessing the service, you'll see the following page:



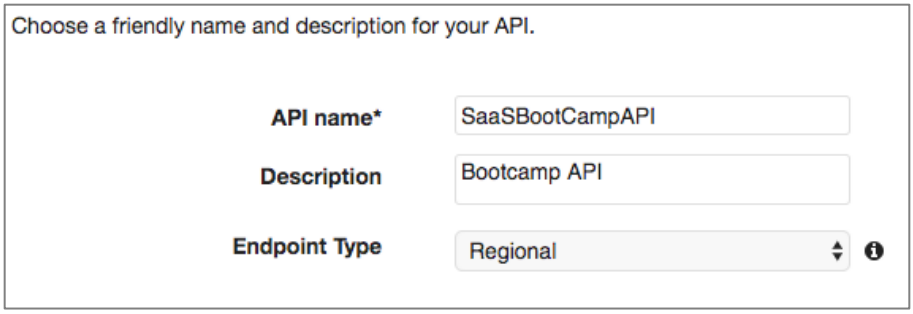
Step 2 – Now select the “Get Started” option. After selecting this button, you’ll be given the option to create an example API. Select the “OK” button from the dialog to proceed.



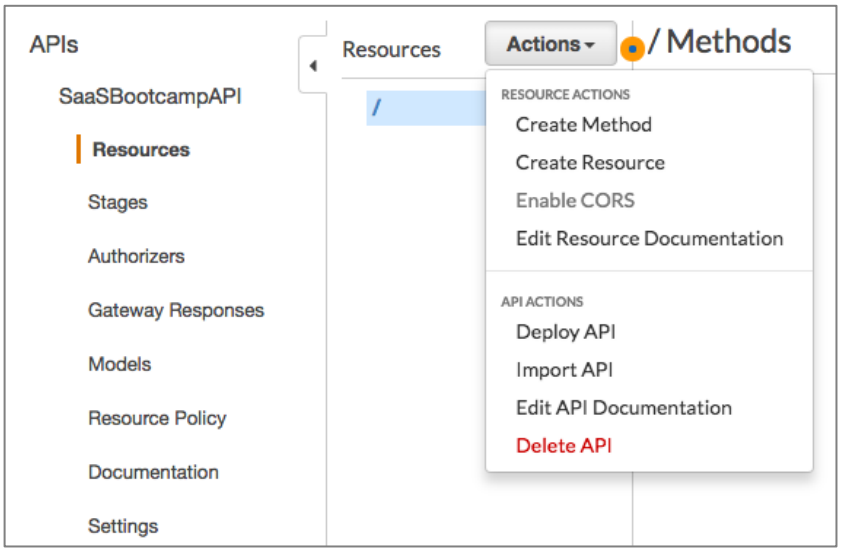
Step 3 – At this point, you’ll be given the option of creating a new API or importing one from Swagger. Select the “New API” option.



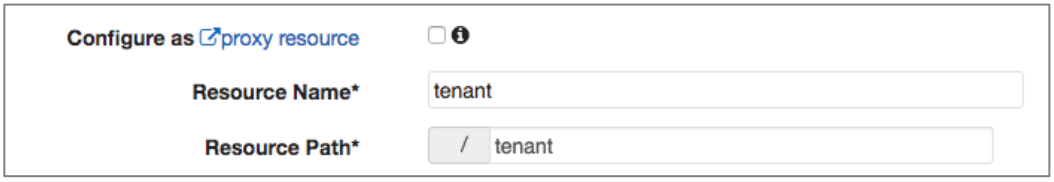
Step 4 – After selecting “New API”, the middle portion of the page will provide a form that is used to name and configure your new API. Enter “SaaSBootcampAPI” for the API name and enter “Bootcamp API” for the description. Then select the “Create API” button.



Step 5 – The API has now been created. We now have to configure the resources and methods that will connect the moving parts of our onboarding flow. To do this, we must first select the root (“/”) item in the resource list. The next step is to add the “tenant” resource since we’ve implemented tenant registration as a child at /tenant/reg. To do this, select the “Actions” pulldown and select the “Create Resource” option.

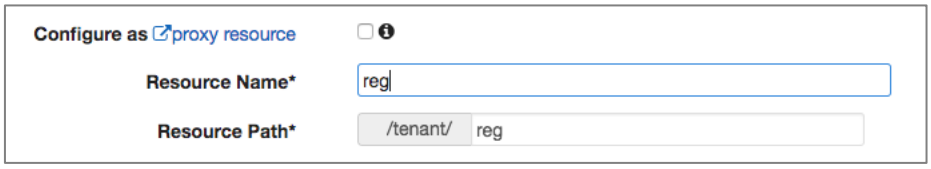


Step 6 – You'll now get the option to enter the name of your resource. Enter the value "tenant" for the name of the resource and select the "Create Resource" button.



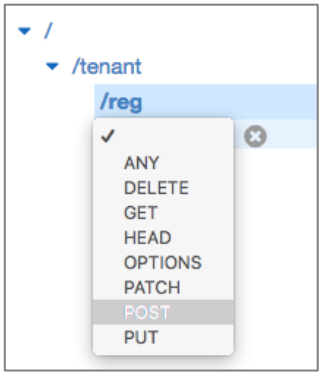
Step 7 – There's one more level we need to add to the resource path we're creating to enable registration. Select the "**/tenant**" resource that we created in the prior step and select "Create Resource" from the "Action" pulldown (as we did in the prior step.

Step 8 – Now you need to supply the name of the resource. As shown in the image below, you need to enter "reg" as the resource name and select the "Create Resource" button.



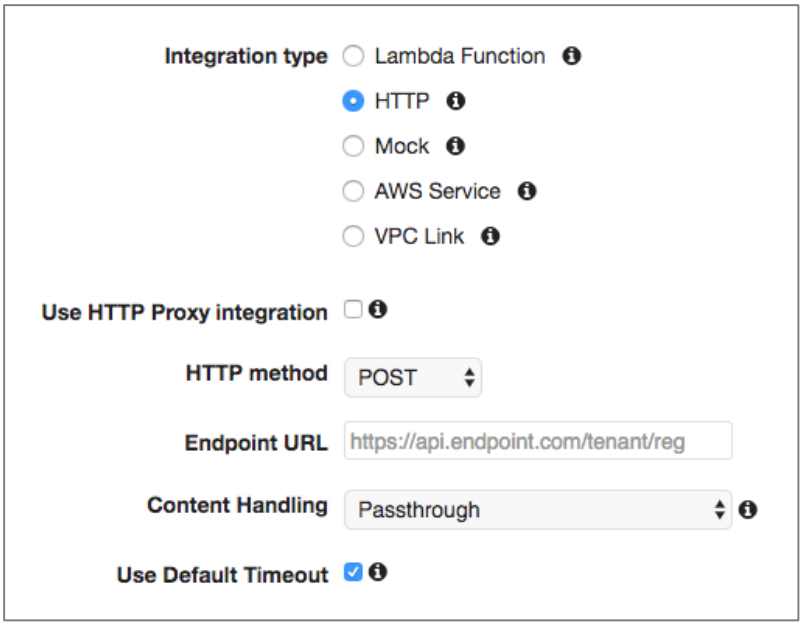
Step 9 – With the "**/tenant/reg**" resource path created, we can implement a specific REST method on this path. In this case, we will be using the POST method as the way to convey that we are registering a new tenant. To create the POST method, you must first select the "**/reg**" resource created in the prior step. Now, select the "Create Method" option from the "Action" pulldown above.

This will add a pulldown list under the "/reg" resource. Pulldown the list and select the POST method (as shown below).



Step 10 – Once you've selected POST, select the checkmark next to the pulldown to complete the creation of the method.

Step 11 – The last step in creating the method is to configure its attributes. First, you'll need to select the integration type. Select "HTTP" for this option. Next you

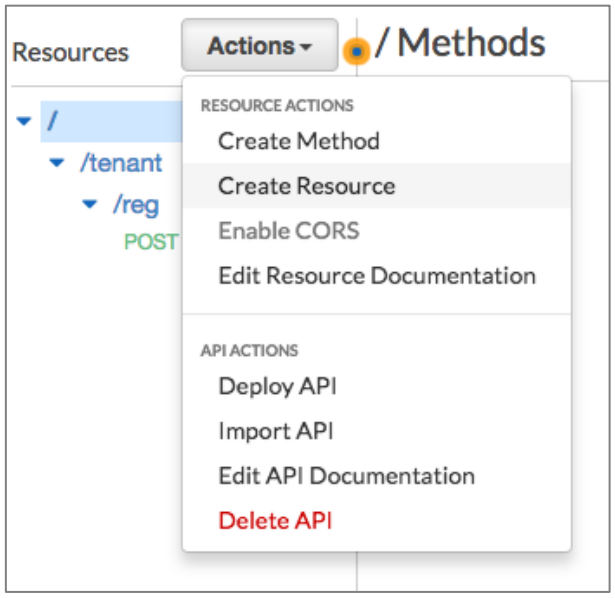


must enter the endpoint URL for the method. We'll use the DNS name A record from the Application Load Balancer which was provisioned prior to the bootcamp.

Step 12 – Open a new browser tab to the EC2 console listed under the Compute heading in the AWS console. Scroll down the left-hand menu and select Load Balancers. In the Description tab you will see the public DNS name we can use to invoke our microservices. For example:

Click the copy icon or select the DNS name with your mouse to copy and return to the API Gateway settings. In the Endpoint URL box, enter **http://LOAD-BALANCER-DNS-NAME/tenant/reg** using the DNS name you just copied.

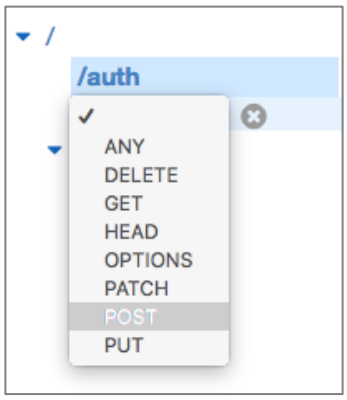
Step 13 – Now we have to repeat this same process for authentication, creating an API Gateway entry point for authentication microservice. To do this, we must first select the root ("/") item in the resource list. The next step is to add the "auth" resource. To do this, select the "Actions" pulldown and select the "Create Resource" option.



Step 14 – You'll now get the option to enter the name of your resource. Enter the value "auth" for the name of the resource and select the "Create Resource" button.

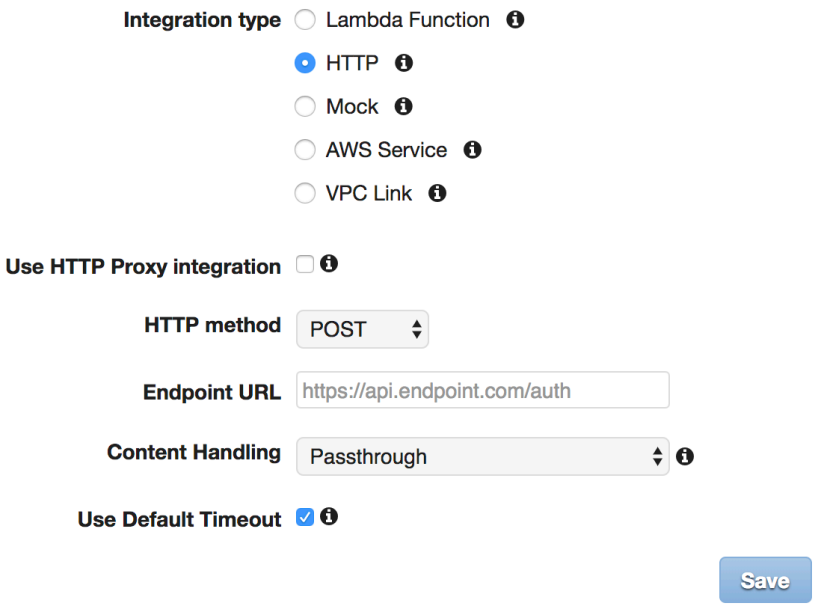
Step 15 – Next, we must configure the methods that we want implemented on the "auth" resource. In this case, we simply need to implement a POST method which will handle our authentication requests. To create the POST method,

you must first select the "/auth" resource created in the prior step. Now, select the "Create Method" option from the "Action" pulldown above. This will add a pulldown list under the "/auth" resource. Pulldown the list and select the POST method (as shown below).



Step 16 – Once you've selected POST, select the checkmark next to the pulldown to complete the creation of the method.

Step 17 – The last step in creating the method is to configure the attributes of the method you just created. Set this method up like you did for the tenant registration method above. Using the same load balancer DNS name you copied earlier as the base URL, enter http://LOAD-BALANCER-DNS-NAME/auth for the Endpoint URL and finish the process by selecting the "Save" button.



Recap: This step was all about creating the wiring between the application client and the system's microservices (registration and authentication). We created two of what will eventually be many API Gateway methods that will support the functionality of our SaaS application.

Part 6 – Deploying the Onboarding & Authentication Application

We've now got the microservices deployed and pieces in place to support the onboarding process. Now we'll deploy the application that can engage the services to onboard and authenticate tenants. We won't dig into the details of the web application here. It's a relatively straightforward AngularJS application that is hosted on Amazon S3.

Step 1 – Before we deploy the onboarding and authentication flows, let's take a look at a sample from the UI code that will be invoking the REST services that we created above. The code that follows is take from the register.js controller found at Lab1/Part6/app/source/client/src/app/scripts/controllers/register.js.

When the registration form is filled out and the user selects the "Register" button, the system will invoke the following snippet of code:

```
$scope.formSubmit = function () {
  if (!($scope.tenant.email || $scope.tenant.companyName)) {
    $scope.error = "User name and company name are required. Please enter these values.";
  }
  else {
    var tenant = {
      id: '',
      companyName: $scope.tenant.companyName,
      accountName: $scope.tenant.companyName,
      ownerName: $scope.tenant.email,
      tier: $scope.tenant.plan,
      email: $scope.tenant.email,
      userName: $scope.tenant.email,
      firstName: $scope.tenant.firstName,
      lastName: $scope.tenant.lastName
    };

    $http.post(Constants.TENANT_REGISTRATION_URL + '/reg', tenant)
      .then(function(data) {
        console.log('Registration success');
        $scope.hideRegistrationForm = true;
        $scope.showSuccessMessage = true;
      })
      .catch(function(response) {
        $scope.error = "Unable to create new account";
        console.log('Registration failed: ', response)
      })
  }
};
```

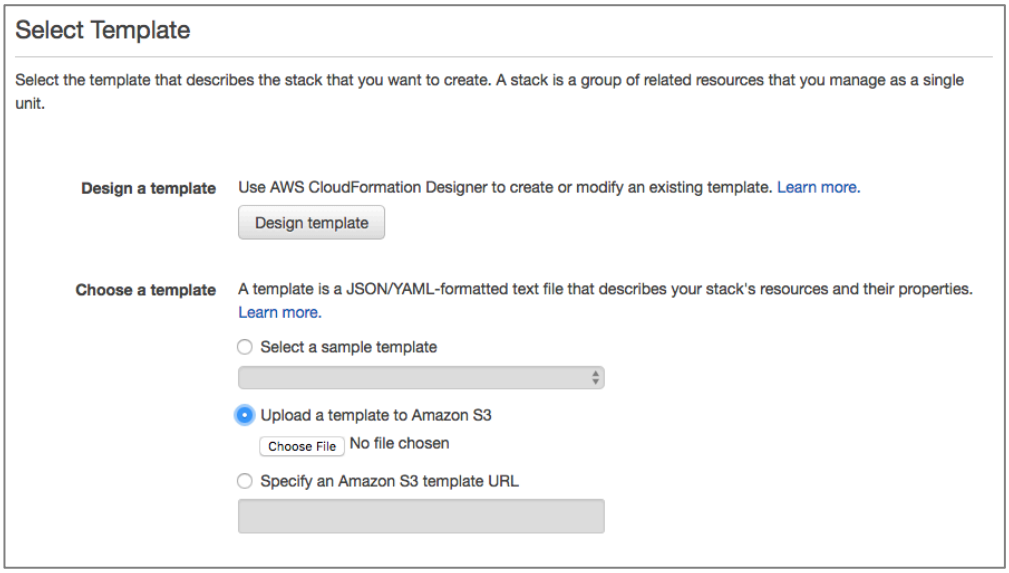
Here you'll notice that we extract the contents of the form and construct a JSON object that holds all the attributes of our new tenant. Then, we make the REST call that POSTs this JSON tenant data to the tenant registration service (introduced above). This registration services then makes calls to the user manager and tenant manager to provision all the elements of the tenant footprint.

Step 2 – Deploying the user interface portion of the application is achieved via a **CloudFormation** template that copies the UI to an **S3** bucket and enables web access to the installed application. To deploy the UI of the application, you'll first need to download the template of **CloudFormation** template, which can be found at Lab1/Part6/templates/application-ui.template.

From here, you need to save the file locally. You can right-click on the "Raw" button in GitHub and select "Save Link As" and name the file "**application-ui.template**".

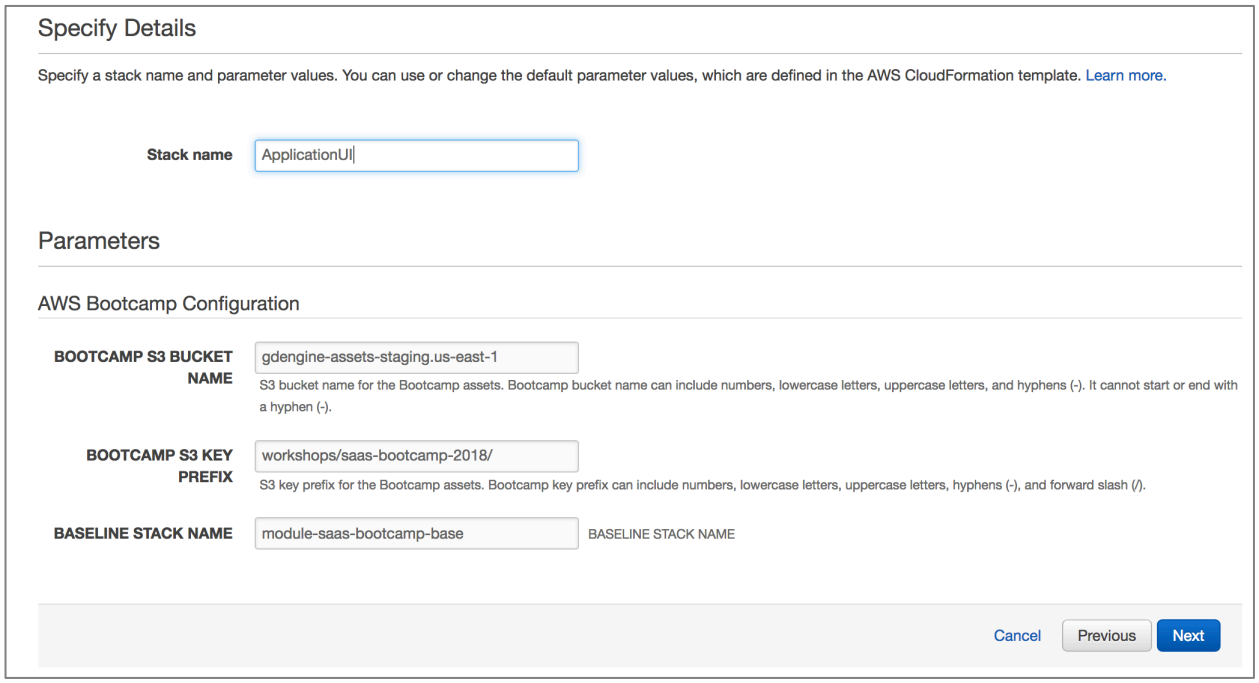
Step 3 – Now we can execute this downloaded template. To start the process, navigate to the CloudFormation service and select the "Create Stack" button at the upper left of the page.

Step 4 – After you have selected create stack you'll be presented with a screen of options for selecting a template to be provisioned. The screen will appear as follows:



Select the "Upload a template to Amazon S3" option. Then, select the "Choose File" button and select the file **application-ui.template** file that was created in Step 2. Finally, select the "Next" button to move to the next step in the creation process.

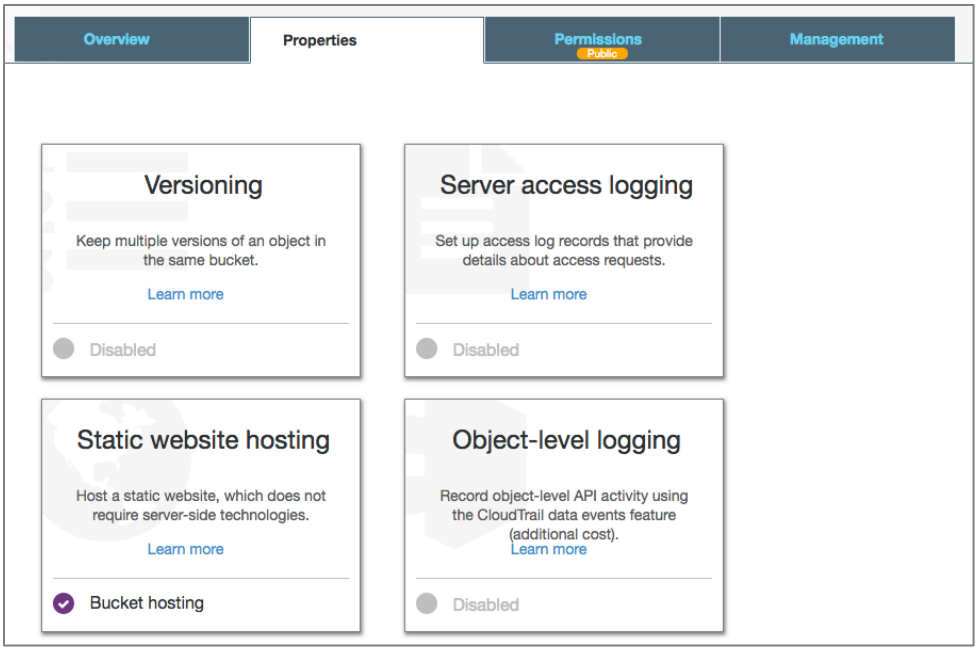
Step 5 – The CloudFormation service will now prompt you for a stack name. Enter "**ApplicationUI**" for the stack name. Select the Next button and the Next button again on the Options page and finally check the box next to the warning that states I acknowledge that AWS CloudFormation might create IAM resources with custom names then the Create button.



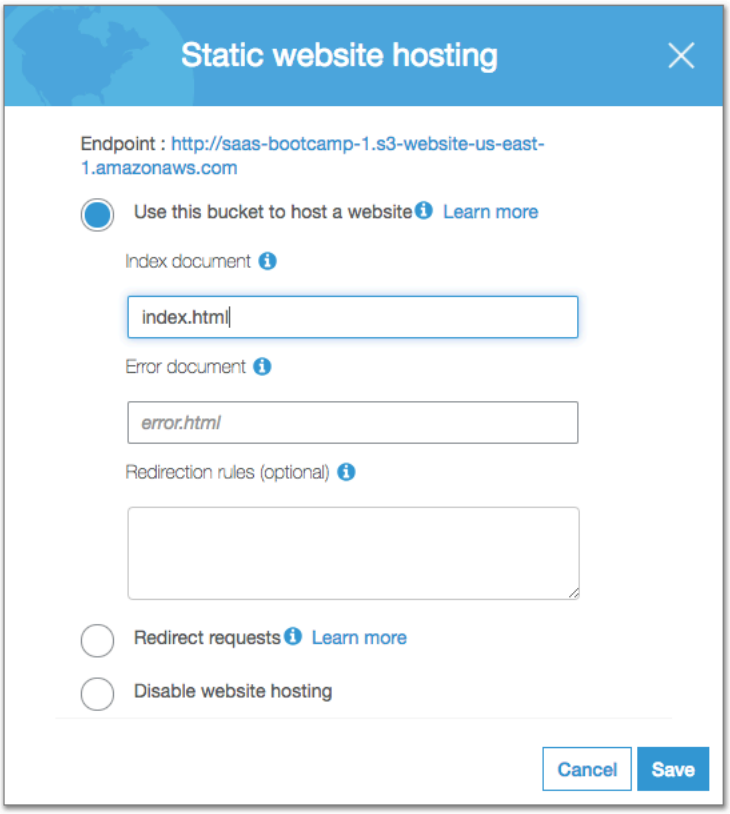
Step 6 – We now have a fully deployed and accessible application. To find the URL for the application, navigate to the **Amazon S3** service and select the link with the bucket name where the application has been deployed. Note, that an **S3** bucket will have been created with a bucket name pattern of **applicationui-webclient-*random*-webbucket-*random*** and it will be the only bucket marked with Public access. For example:

Bucket name	Access
applicationui-webclient-aujf7okllc-artifactbucket-fi2kgnsd21z8	
applicationui-webclient-aujf7okllc5y-webbucket-1ihak2ib6br87	
applicationui-webclient-d39toyxiph-artifactbucket-j7xhownuhh6q	Not public *
applicationui-webclient-d39toyxiph3v-webbucket-1bd14i4hf62i9	Public
cf-templates-ev9wfyeehhpn-us-east-1	Not public *
module-saas-bootcamp-base-base-destinationbucket-k8ov3b1wrw8k	Not public *
module-saas-bootcamp-base-base-t77-artifactbucket-lxrj04125w2n	Not public *
module-saas-bootcamp-base-base-t77q8i2w-keybucket-z1h0alcfr7gb	Not public *

Step 7 – Once you're in the detailed view of the bucket, you'll see a list of tabs across the top of the screen. Select the "Properties" tab and you'll see the following options displayed:



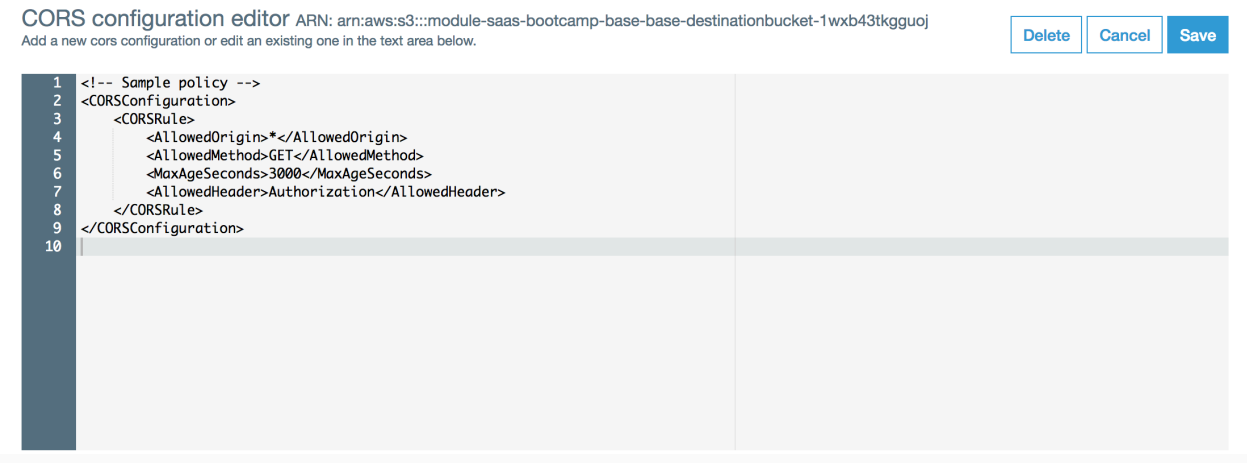
Step 8 – Now select the "Static website hosting" option and you'll see the URL endpoint for your application. Choose the "Use this bucket to host a website" option.



Step 9 – Enter "Index.html" as the value for the "index document". Also, capture and record the **endpoint URL** at the top of this form. It represents the entry point of your application and we'll be using it in subsequent steps. Finally, select the "Save" button to complete the configuration of your bucket.

Step 10 – Navigate to "Permissions," and click on "CORS" configuration, and ensure that the following bucket policy is configured.

```
<CORSConfiguration>
   <CORSRule>
      <AllowedOrigin>*</AllowedOrigin>
      <AllowedMethod>GET</AllowedMethod>
      <MaxAgeSeconds>3000</MaxAgeSeconds>
      <AllowedHeader>Authorization</AllowedHeader>
   </CORSRule>
</CORSConfiguration>
```



Step 11 – Use the URL to access the application with your browser.

Recap: We now have a UI deployed to **S3**. This was achieved by simply packing up an AngularJS application and deploying (with its related assets) to an **S3** bucket and opening up that bucket for public access as a website (a standard **S3** feature).

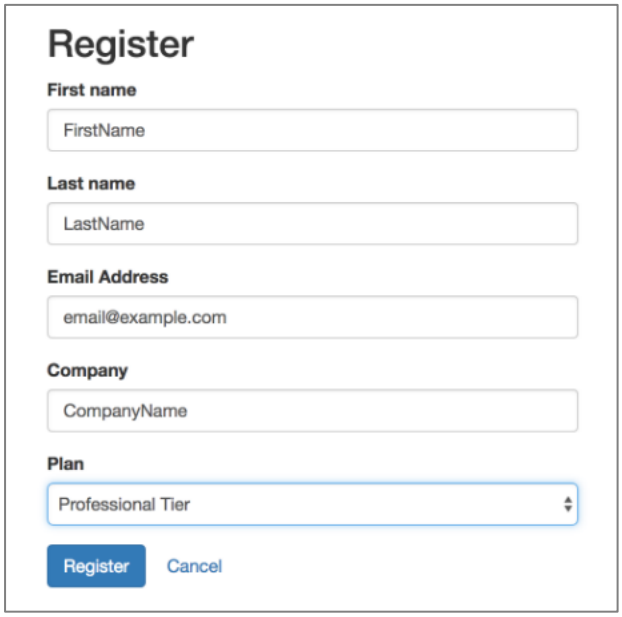
Part 6 – Onboarding A New Tenant with the Application

Finally, we've reached the point where all the elements of the UI, the backend services, and the supporting AWS services have been configured to enable the onboarding of a new tenant. It's now time to see all of this play out via the complete onboarding experience. The steps that follow will take you through the experience of registering a new tenant through the eyes of the end user.

It's important to note that many of the rules and mechanics of this onboarding reflect the realization of the policies that were created in the prior configuration of Cognito. Validation mechanisms and password policies, for example, will be enforced and orchestrated by Cognito.

Step 1 – This whole process starts, as you might imagine, by entering the URL of our application into the browser. This is the URL from our S3 website that's hosting the static assets of our application. We captured this URL from the S3 bucket configuration above.

Step 2 – Upon displaying the landing page for the application, you'll be prompted to login. This page is for tenants that have already registered. You don't have any tenants yet, so, you'll need to select the "Register" button (to the right of the "Login" button). Selecting this button will take you to a form where you can register your new tenant.



Step 3 – Enter the data for your new tenant. The key value here is your email address. You must enter an email address where you can access the messages that will be used to complete this registration process. The remaining values can be as you choose. The "Plan" value here is simply included to convey that this would be where you would capture the different tiers of your product offering. Once you complete the form, select the "Register" button and you'll be presented with message indicated that you have registered and should be receiving an email to validate your identity.

Step 4 – 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 5 – 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 6 – Enter the temporary credentials that were provided in your email and select the "Login" button.

Step 7 – 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, the 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 8 – After you've successfully changed your password, you'll be logged into the application and land at the home page. We won't get into the specifics of the application yet.

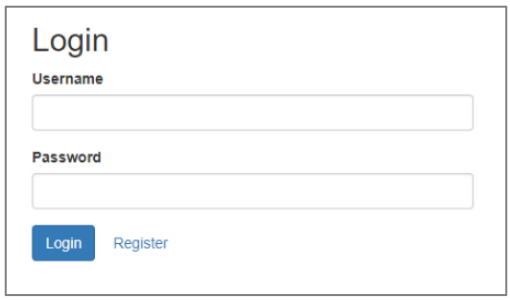
Recap: At this point you've demonstrated that you have all the elements in place to provision a new tenant in your system. This includes the creation of the tenant's identity, the creation of a tenant admin user, the configuration of user attributes, and the setup of policies that impact onboarding, authentication, and the tenant lifecycle. This is a substantial milestone in the creation of your application and lays the foundation for a more seamless approach to authenticating and conveying user/tenant identity across the rest of your SaaS system.

Part 7 – Authenticate a Registered Tenant

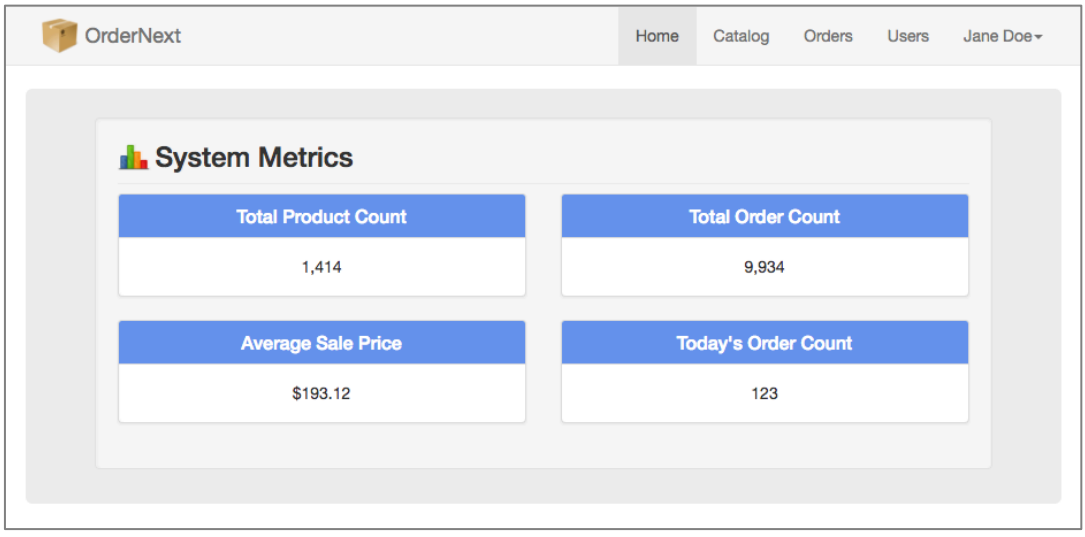
The last step in this process is to confirm that you can authenticate with your newly created account. This is simply about verifying that the data flow of the system works.

Step 1 – Navigate to the application home page by entering the URL of your application into the browser (captured above).

Step 2 – You will be prompted to login to the application. The following page will be displayed:

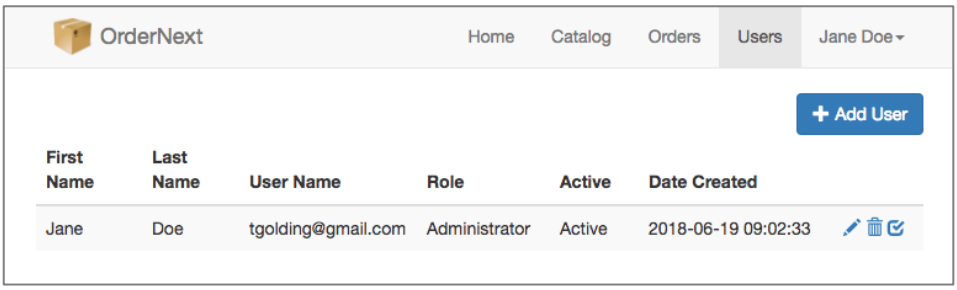


Step 3 – Enter your username (email address) and the password you created during the initial login above. You will now be placed at the landing page of the application (shown below).

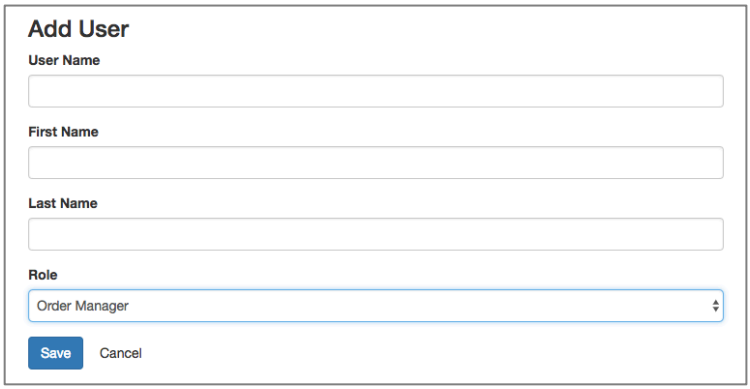


Step 4 – As a new tenant to the system, you are created as a "Tenant Administrator". This gives you full control of your tenant environment. It also gives you the ability to create new users in your system. Let's try that. Navigate to the

"Users" menu option at the top of the page. This will display the current list of users in your system.



Step 5 – Now select the "Add User" button to add a new user to your system in the "Order Manager" role. Create the new user (using a different email address/username) than your tenant. Be sure to select the "Order Manager" role. Once you've entered all the data, select the "Save" button at the bottom of the form. Hint: you can use the same email address as you used for your tenant admin but add a plus symbol and unique string after the username and before the @ symbol (e.g. test@test.com -> test+user1@test.com). The email server should also deliver this message addressed to test+user1 to the test user's inbox.



Step 6 – The onboarding of new users follows the same flow that was used to register a new tenant. Pressing the save button triggers Cognito's processing to generate an email that will be sent to the email address that was provided.

Step 7 – Retrieve the email with your temporary credentials.

Step 8 – Logout of the application by selecting the dropdown with your tenant name at the top right of the page. When you select the pulldown, you'll be given the option to logout of the select. Select "Logout" and you will be returned to the login page of the application.

Step 9 – Now, login to the application with the temporary credential you were provided for your new user.

Step 10 – You'll be prompted to change your password. Create a new password and select the "Confirm" button.

Step 11 – This drops you into the application. However, you're now in the application with the role of "Order Manager". You'll notice that the "Users" option is gone from the menu and you no longer have the ability (for this user) to create or manage users in the system.

Recap: This was the last step in verifying that all the elements of the onboarding and authentication lifecycle are in place. We logged back into the system as our tenant admin user and verified that the newly setup password would let us into the application un-challenged. We also created a user as a child of our tenant and saw that the onboarding flow was the same and that the application restricts access to certain functionality not only by the type of user (tenant vs user) but also via the custom attributes we defined in Cognito such as role and tier.