# Part 5: How To Build a Serverless + MicroService for your Application

*Author:* [*David P. Lopez*](http://www.DavidPLopez.com)

*Publisher:* [*UniqueSoftwareDevelopment*](https://www.uniquesoftwaredev.com)

This is a continuation of our multi-part series on building a simple web application on AWS using AWS Lambda and the ServerlessFramework. You can review the first and second parts of this series starting with the setup of your local environment at:

* [Part 1: How To SetUp Your local Serverless Environment](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToSetUpYourLocalServerlessEnvironment.md)
* [Part 2: How To Configure Your Serverless Backend API](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigureYourServerlessBackend.md)
* [Part 3: How To Configure Your Infrastructure As Code, Mock Services, & Unit Testing](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigure.IAC.Mocks.UnitTests.md)
* [Part 4: How To Deploy and Configure an effective CI/CD Pipeline on AWS](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToReviewServiceToConfigureCICDpipeline.md#part-4--code-review-deploy--configure-an-effective-cicd-pipeline-on-aws)

You can also clone a sample of the application we will be using in this tutorial here: [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api)

Please refer to the repo above as you follow along with this tutorial. In this part of the series we will cover implementation steps for *User Authentication* with AWS Cognito and the implementation of the application’s *Data Model* using AWS Dynamo DB as *serverless + microservices* that we will deploy as *Infrastructure As Code* on AWS CloudFormation. The *business logic* that will govern the functionality of this application will run on AWS Lambda to allow us to implement a simple General Ledger that we can use to think about simple problems that brought about the rise of Blockchain technology.

We will start with the [ServerlessStarterService](https://github.com/lopezdp/ServerlessStarterService) that we implemented in a previous chapter in this series, and we will extend this demo and turn it into the [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api) discussed above.

## System Architecture & Project Structure

We will start by structuring our project directory according to what we will build today. I have done some work in the *payments* industry, and I always enjoy looking for ways to make it easier to send or receive money between people. In my opinion, *sustainability* and *monetization* are synonymous. You cannot sustain any of your projects without Cash Money. We will build a *time-sheet* application that will allow a user to login to the demo application, create an invoice with a description of work, to send it to an email to notify a *third party* and accept payment for the invoice generated. We will call this the **PayMeNow** application that will use a DynamoDB table as its General Ledger.

You can [clone the invoice-log-api](https://github.com/lopezdp/invoice-log-api) that we are starting out with if you want to see the completed application, but we think it is best if you just follow along with us through each step, so you can better internalize the lessons to be learned in this tutorial my young *padawan*. Let me know when you get sick of the *Star Wars* references and we will double down a bit more. Go ahead and start with the [ServerlessStarterService](https://github.com/lopezdp/ServerlessStarterService) and let’s do this:

We are going to continue on, where we left off in [Part 2 of Setting up your local](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigureYourServerlessBackend.md#setup-serverless-framework-locally) serverless environment. We walked through creating a local project structure that resembled something like this after you renamed your template:

PayMyInvoice  
 |\_\_ services  
 |\_\_ invoice-log-api (renamed from template)  
 |\_\_ FutureServerlessMicroService (TBD)

Proceed to navigate into the invoice-log-api service where we will now implement a few Lambda functions that we need to serve our [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api). The first thing a user of our application will need to do is to create an invoice that the user can send to someone. The recipient of the invoice will make a payment to our system based on the amount due on the invoice. After having initially implemented this service to be able to more effectively write this tutorial for you, I now realize that I have forgotten to add fields for Payee information and email which would really be helpful within the scope of this application! Either way, my mistake is your success!

### Implement createInvoice.js

Here is what you will need to implement to be able to let your users create an invoice that they can use to accept payments for from a third-party within our new payment system. I have left a few comments within the source code to provide some insight on what each line of code is trying to achieve. We will review these in a bit of detail to be sure you are comfortable implementing a few Lambda’s in the future on your own!

**creatInvoice.js**

// Step1: Discuss the uuid import and need for uuid  
import uuid from "uuid";  
// Step2: Discuss in tutorial why /lib and responseLib  
// exist and why they are used.  
import { success, failure } from "./libs/responseLib";  
  
// Step3: Discuss in tutorial the need for a Dynamo table  
// and a dynamo table implemented for each microservice  
// Implement service below  
import \* as dynamoLib from "./libs/dynamoLib";  
  
export async function main(event, context) {  
 // The request body is passed in as a JSON string  
 // in event.body and it needs to be parsed!  
 const data = JSON.parse(event.body);  
  
 // Now you need to store the information into an  
 // object that you will use later to store the data  
 // collected from your user into a database of invoices!  
 const params = {  
 // Need to describe in tutorial how this object is  
 // accessing the table defined in the serverless.yml  
 // and how data is being stored in db.  
 TableName: process.env.invoiceTbl,  
 /\*  
 \* 'Item': contains the ttributes of the item to create  
 \* In NoSql this is equivalent to a row of data  
 \*  
 \* - 'userID': user identities are federated through  
 \* the CognitoIdentity Pool (MUST create),  
 \* and we need to use an identityID as the  
 \* userId of the Authenticated User.  
 \*  
 \* - 'invoiceID': a uuid that is unique to this record  
 \* - 'createdAt': this is the current UNIX timestamp  
 \* - 'payerEmail': this is the email of the invoice Payer  
 \* - 'description': this is the description of the  
 \* transaction as entered by the user  
 \* and parsed from the Request.body  
 \*  
 \* - 'amount': this is the amount of the transaction  
 \* as entered by the user and parsed from  
 \* the Request.body also  
 \*  
 \* - 'attachment': This any file the user may attach  
 \* to the record that the system will  
 \* store in an s3 bucket and parsed  
 \* from the request.body too  
 \*/  
 Item: {  
 // Discussion on DynamoDB sort key design and  
 // how to properly aggregate data in table to  
 // minimize READS (RCU) from cloud to minimize cost!!!  
 userId: event.requestContext.identity.cognitoIdentityId,  
 invoiceId: uuid.v1(),  
 createdAt: Date.now(),  
 description: data.description,  
 amount: data.amount,  
 attachment: data.attachment  
  
 }  
 };  
  
 try {  
 // async call to db??????  
 // need to implement and test!!!!  
 // need a lib to make calls to aws dynamo  
 // to persist data for new invoices created  
 await dynamoLib.call("put", params);  
 return success(params.Item);  
 } catch ( err ) {  
 return failure({  
 status: false  
 });  
 }  
}

We’ll break this first one down by steps just to be sure this is all clear to you; *Step1* is where we import the functionality needed to allow us to generate a new uuid for each invoice created. The uuid will provide us with the ability to support cross-platform, UUID versions 1,3,4, and 5 that use *cryptographically-strong* random numbers with *zero-dependencies*. uuid should be listed as a dependency in your package.json if you cloned the [ServerlessStarterService](https://github.com/lopezdp/ServerlessStarterService). If you are implementing this from scratch then you will need to run: $ npm install --save uuid. In this application we are using uuid.v1() as the *uniqueId* for each invoice the user creates.

#### CORS (Cross-Origin Resource Sharing)

In *Step2* we had to create a library that we located at: ./libs/responseLib. Or, one level up in your project directory, under your new /libs directory. We are implementing this to be able to send consistent responses back from our dynamoDB resources, as invoice objects making use of the appropriate http status codes. Each service will have to respond with the correct statusCode and response headers so that our resources can be shared accros our *serverless + miscroservices*. This is known as **CORS (Cross-Origin Resource Sharing)**. With our dynamoDB implementation, we will have to respond with statusCode: 200 if our http requests are successful. Otherwise, the response from our *serverless + microservice* will be a statusCode: 500. We are using the ./libs/responseLib library to attempt to keep ourselves **DRY**!

#### JavaScript Promises to Love You

In *Step3* we implement an import that uses a library that we call ./libs/dynamoLib, in another attempt to *save the world* by decoupling our code and making it more *modular* and easy to read! Our goal here is to create a JS Promise library that we can use to make our code super simple minded for *cavemen* like me.

We just want a way to replace the standard syntax for the JavaScript callback functionality. If you have a better way to manage asynchronous code then please *HMU*, otherwise, this tutorial is going to make do with JavaScript *Promises*. The beauty behind using these *Promises* with our async/await pattern that we show in the Lambda implementation above, is that we can just return our response as soon as our service completes the execution of its logic, which allows us to avoid using the callback. If you’ve been using a callback since 2012, and it’s what you know, then you may disagree. Please, send me the blog post you write about it telling me how I am wrong, and make sure to scream at me on Twitter. We’re just going to push ahead with all the cool stuff **ES6 Syntax** keeps on giving us.

**Another Promise from DynamoDb**

// Need to use and import the aws-sdk  
import AWS from "aws-sdk";  
  
AWS.config.update({  
 region: "us-east-1"  
});  
  
export function call(action, params) {  
 const dynamo = new AWS.DynamoDB.DocumentClient();  
  
 // return a promise with the results for  
 // the specified action and params  
 return dynamo[action](params).promise();  
}

## Serverless + MicroService & the DynamoDB *DataStore*

Coming up in this next section, we will also have to define the *NoSQL* tables that we will need to implement in AWS DynamoDB within an isolated environment, so that we can be sure to develop the appropriate data model needed for this specific *serverless + microservice*. Furthermore, in a *microservice* environment, each *service* will implement its own *database*. In the case of *NoSQL*, its own table (more on this to come).

The idea behind a *microservice* based architecture, for those of you not already in the know, is to be able to more easily maintain your code, and extend an infinite amount of features that you believe will *save the world*, in a *decoupled* environment that will allow you to build out new functionality without impacting the work of your team implementing their own versions of this application. Simply put, we just want to write code and develop services that only deal with one specific thing or task.

We will use a loosely coupled architecture that makes it easy to develop, test, and deploy new features independently of each other, and to maintain more control over the stability of the system. No two services should rely on data from each other or any other source, nor should they know anything about the other’s state. In a *serverless + microservice* environment, each *microservice* is going to have its own database, that will deal with the specific attributes that the service in question needs, to provide the correct response to any given request, while using the data it persists to its own data store.

We’ll get to the implementation of our DynamoDb tables soon enough *Danial-san*. You’ll need to show me that you can still make them #AlphaMoves though, so take it easy and let’s just take this one step at a time. For now, you’ll need to get back to [*Paint The Fence*](https://www.youtube.com/watch?v=R37pbIySnjg).

**Get To Work**



We will need a new directory that we will call resources. In the root of your *serverless + microservice* project. Proceed to $ mkdir resources, so that we can have a place to save the definition of the DynamoDb tables that we are going to use for our new *B2B* PayPal clone that we call, **PayMyInvoice**. Inside of your new resources directory that you have created for your *serverless + microservice*, I will need you to go ahead and create a new file called GeneralLedgerTable.yml. Below is a gist of what you will need to implement in this new file of yours [Bud](https://www.youtube.com/watch?v=6CMZSw7cS8M):

“Life all comes down to a few moments… This is one of them.” - *Bud Fox, WallStreet*

**DynamoDB without a Server!!!**

# NOTE: DynamoDB Serverless Configuration  
  
Resources:  
 GeneralLedger:  
 Type: AWS::DynamoDB::Table  
 Properties:  
 TableName: ${self:custom.tableName}  
 AttributeDefinitions:  
 - AttributeName: userId  
 AttributeType: S  
 - AttributeName: invoiceId  
 AttributeType: S  
 KeySchema:  
 - AttributeName: userId  
 KeyType: HASH  
 - AttributeName: invoiceId  
 KeyType: RANGE  
  
 # Set the capacity based on the stage  
 ProvisionedThroughput:  
 ReadCapacityUnits: ${self:custom.tableThroughput}  
 WriteCapacityUnits: ${self:custom.tableThroughput}

The first thing we define here under the Resources declaration is the name of this table which we have appropriately chosen to call our GeneralLedger. The actual table that we implement and get back from this CloudFormation template is named after a custom variable that we call: ${self:custom.tableName}. Our serverless.yml will generate this table for us dynamically within AWS.

We are also configuring two of our table’s field attributes that we have called userId and invoiceId as shown in the example. We complete the implementation of our table by provisioning the read and write capacity of our database using custom variables to describe the load that our tables will need to service as we attract users to our application over time. More on this subject and autoscaling capacity later, for now just try to understand what we’re doing.

### Add a DynamoDB Resource to your CloudFormation template

Going back to the serverless.yml file that we have in the root of our project directory, now I need you to add the following reference to our GeneralLedgerTable.yml, as a resource to this project. You will have to replace the resources block that is at the very bottom of our serverless.yml file. Use the information below to make the adjustments that we need to make:

**Add a DynamoDB Resource**

# Keep resources modular and create each with separate CloudFormation templates  
  
resources:  
 # DynamoDB Services  
 - ${file(resources/GeneralLedgerTable.yml)}

There are a few considerations we need to make while studying the implementation I have just graced you with. The first and most important thing I would ask you to seriously consider is to **STOP THINKING RELATIONALLY**! This is a NoSQL data model and if you try to build a *Relational Database* out of it, you’re going to engineer yourself right on out of house and home. Consider it a [NoSQL Best Practice](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html) to just shove in as much of your data into one table as possible. Therefore, the only thing that you really have to declare and think about ahead of time is a couple of concepts you need to know surrounding Composite Keys; Namely, your *NoSQL* Partition Key and SortKey.

Please pay attention. We will take the liberty now to go off on a bit of a tangent here to discuss a few of the fundamentals surrounding the *Magic* that are the **M**assively **A**ggregated **D**ata models that we now know as DynamoDB. Something with more power than the *Cold War* era architects of Mutually Assured Destruction could ever have imagined. This whole [**DARN**] thing is just **MAD**.

The lesson to learn is that it does not matter how bad it gets, the only way to become a real *professional* is to realize that it is all a mess and that our job as *Software Engineers* is to figure out some way to help our organizations achieve their goals and objectives. That is all we get paid to do; We get paid to implement the ideas of those who are in charge. If you ask me, this is where the greatest opportunities reside.

Learn DynamoDB and become an expert at its [Best Practices](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html), the quicker you can learn to adapt to changing market conditions, the better of an engineer you will become for it.

### DynamoDB Key Types

Key Types determine how your application can access the data it collects later on. There are two Key types you can use to define for your table, furthermore all Key Type Attributes **MUST** be decided upon in advance. We can use either SimpleKey or CompositeKey types. To take advantage of the Distributed Hash Map Architecture that enables DynamoDB’s high performance as a Key:Value *Document Storage* database, we will use a CompositeKey.

The simplicity that DynamoDB provides you with is that it is *Schemaless* in that it does not require you to define every field you need for this service ahead of time. The only two fields we do need to declare now, however, are as follows:

1. **Partition Keys**: These will uniquely identify a *Partition* of records that you will have stored in your *NoSQL* table. In our case we have called our table /resources/GeneralLedgerTable.yml. There are very few use cases that justify multiple tables. In this new reality you want to implement **ONLY ONE TABLE** that is going to partition our data in a distributed fashion so that we can sort through out data efficiently.
2. **Sort Keys**: This **Key** will have a **Value** that will differentiate our Items within a distributed *Partition* that persists data to our *document storage* system. This **Sort Key**, also known as a RANGE key, will be combioned with our **Partition Key** to let DynamoDB use it to catalog our information within its data store according to the relationship of the Item to its partition:sort *composite key*. This **Sort Key** will allow us to *Sort* the data stored within a given **Partition** so that we can filter out our *Items* using a specific set of filters and conditions. Later we will show you the tools that DynamoDB gives you to create different **Access Patterns** that will let your sort your data and the **Items** in your *document storage* system in different ways.

The thing to remember about *Composite Keys* is that all your *Items* are stored together if they share a *Partition Key*. Each Item is sorted within this *Partition*, and sorted within the DynamoDB physical storage system using the value of its *Sort Key*.

*Sort Keys*, if designed correctly will allow you to eliminate complex JOIN statements in exchange for *Composite Keys* that allow you to query *Composite Data* that you will aggregate into one table. The idea is to keep related data together under the roof of one *serverless + microservice*, to create aggregated tables that allow you to create **views** of the data you collect from the user. To accomplish this, DynamoDB gives you a couple of tools to help you address *Complex Queries* that you will have to solve for to build an application that your users will want to use.

“Build Something that people want!” - YCombinator

### DynamoDB Indexes

In a NoSQL **Data Model** you need to avoid thinking in a **Relational** manner because it will cost you more money due to the amount of **Read Requests or RCUs** you will make to your database. DynamoDB does not enforce relationships between tables. As you aggregate composite data into your NoSQL DynamoDB implementation, your data is stored as [*unnormalized*](https://en.wikipedia.org/wiki/Unnormalized_form) information. If your application cannot tolerate showing or outputting *stale* data to your user, then you need to rethink using DynamoDB and reconsider using a *Strongly Consistent* **RDBMS** like PostgreSQL instead.

We can take advantage of DynamoDB when *stale* data is not an issue, and when *Eventually Consistent* data is acceptable for your use case. In practice, data is processed by AWS DynamoDB so fast that your implementation will be very close to, if not *Instantly Consistent*. The idea is to sacrifice strong consistency in exchange for a highly efficient *document store* on a distributed hash map that is schemaless and easy to implement, to achive high availability so that every request made to your database receives a successfull, *non-error* response. DynamoDB is highly scalable due to its efficient partitioning mechanism that distributes your data across a series of highly available nodes of data stores that can also enable *Realtime Operations*. DynamoDB can update tables across your services with **Sub-Second Latency**. DynamoDB will also enable you to process *sharded data* within your application to process *streams* of updates to your database in **Realtime** at a very **low cost**.

**It is all about the Benjamins**

In forcing you to declare and define your *Partition* and *Sort* key attributes ahead of time, DynamoDB requires you to define the **Access Patterns** that your application will need to implement to query your database, and its *schemaless* data store, before you start using it. With DynamoDB you will normalize your data as you query your data store. You will generate your view and normalize your data as you scan or fetch the information you need from your database.

Ideally, you will aggregate all of the information your application collects, and you will create different views of original data with the models that you define. Your queries will deaggregate the data you gather, to implement your features, as they stream your data in Real Time to your application. More importantly, to generate the views that you will output to your users, DynamoDB will force you to start by **defining the questions that your application needs answered with your queries** first!

Keeping related data together in each service will let you define the **Access Patterns** needed to use your *Composite Primary:Sort Keys* effectively so that you can distribute your queries evenly across your NoSQL partitions. To accomplish this DynamoDB give you the following tools to better manipulate and stream your data to your *frontend* views:

1. **Local Secondary Indexes**: These are similar to your *Sort Key* types in that you can define up to 5 **LSI**’s to re-Sort your queries within the same *Partition*, as needed. Defining an additional *Sort Key* in the form of an **LSI** gives you the ability to find the information you need in your database with a different set of **Access Patterns** depending on the view that you need to display to your user. You would use an **LSI** to re-Sort the results of a query with a different field or attribute that you must define ahead of time.
2. **Global Secondary Indexes**: A **GSI** is another tool that DynamoDB gives you to query your data with more flexibility and ease. A **GSI** is nothing more than a copy of your table with a different **Partition Key** and **Sort Key** that gives you the ability to store a subset of attributes while emulating the functionality of an **LSI**. DynamoDB lets you define up to 20 **GSI**’s to give you a flexible and *eventually consistent* view of your data that can be *unlimited* in size. More **GSI**’s give you the ability to ask more questions of the data you store in your table.

The difference between these tools provided to you by DynamoDB, is that you will want to use each of them based on the needs of your queries, or the **Access Patterns** you define when modeling your data. You will want to use either your **LSI** or your *Sort Key*, but not both. Use either, depending on the conditions you present in the questions that you determine your application will need to ask of your *data store*. Use a predefined **GSI** when your application needs to display a view that relies on a completely different *Partition* of data that will need to be obtained with a different **Access Pattern**. DynamoDB is a highly scalable tool that gives you a lot of flexibility to quickly implement and iterate through our application. Now that you understand how to model your data, let’s keep implementing the [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api) application.

### Provisioning Table Throughput Capacity

DynamDB also helps us balance our costs against the availability of our database by letting us autoscale our database to meet the needs of our users. Using the settings in this section will need careful consideration on your part because these can lead an unexpected surge in your AWS Costs if your application goes viral. These settings will allow your database to scale to meet the increasing demand of users on your application. Keeping these settings static would prevent your database from responding to queries that exceed an arbitrary threshold. Instead, we will let it grow with the needs of our users. We want to register as many users as our application can handle and we want it to scale with the demands of our market.

“You have to spend money to make money.” - *Anonymous*

1. **Provisioning Throughput (RCU vs WCU)**:

Let’s adjust the definition of our GeneralLedgerTable that we created above so that it looks a little more like this:

# NOTE: DynamoDB Serverless Configuration  
  
Resources:  
 GeneralLedger:  
 Type: AWS::DynamoDB::Table  
 Properties:  
 TableName: ${self:custom.tableName}  
 AttributeDefinitions:  
 - AttributeName: userId  
 AttributeType: S  
 - AttributeName: invoiceId  
 AttributeType: S  
 KeySchema:  
 - AttributeName: userId  
 KeyType: HASH  
 - AttributeName: invoiceId  
 KeyType: RANGE  
 # Set the capacity based on the stage  
 # ProvisionedThroughput:  
 # ReadCapacityUnits: ${self:custom.tableThroughput}  
 # WriteCapacityUnits: ${self:custom.tableThroughput}  
 # AWS DynamoDB Automatic Provisioning Pricing  
 # https://medium.com/@softprops/putting-dynamodb-scalability-knobs-on-auto-pilot-3af8520439c9  
 BillingMode: PAY\_PER\_REQUEST

If you notice, you will need to comment out every thing from # ProvisionedThroughput: down and then you will need to be sure to add:

BillingMode: PAY\_PER\_REQUEST

This will tell CloudFormation to configure your DynamoDB to autoscale your tables when the need arises. AWS will bill you for the demand placed on the the service alone and nothing more. When peak load decreases, so to will the resources allocated to your service. Collecting this information will help you determine what the best approach will be in the future when considering which configuration options are best for you and your company’s situation.

### Configure DynamoDB as IAC on serverless.yml

Next, we need to configure our **Infrastructure As Code** template that we are using to tell CloudFormation how to build and deploy our resources and technology stack on AWS. Navigate into the root of your project directory and open up the $ cd ~/services/invoice-log-api/serverless.yml file, and add your DynamoDB configuration information as shown below. Make sure your custom block looks like the version below to tell CloudFormation how to deploy the DynamoDB resources that we will need:

**Serverless.yml custom block**

# configure plugins declared above  
custom:  
 # Stages are based on what is passed into the CLI when running  
 # serverless commands. Or fallback to settings in provider section.  
 stage: ${opt:stage, self:provider.stage}  
  
 # Set your table name as needed for local testing  
 tableName: ${self:custom.stage}-invoices  
  
 # Comment out these settings because we are now on AUTOSCALE  
 # and we are using the BillingMode: PAY\_PER\_REQUEST setting  
 #  
 # Set our table throughput for prod & dev stages  
 # tableThroughputs:  
 # prod: 5  
 # default: 1  
 # tableThroughput: ${self:custom.tableThroughputs.${self:custom.stage}, self:custom.tableThroughputs.default}  
  
 # Load webpack config  
 webpack:  
 webpackConfig: ./webpack.config.js  
 includeModules: true  
  
 # ServerlessWarmup Configuration  
 # See configuration Options at:  
 # https://github.com/FidelLimited/serverless-plugin-warmup  
 warmup:  
 enabled: true # defaults to false  
 folderName: '\_warmup' # Name of folder generated for warmup  
 memorySize: 256  
 events:  
 # Run WarmUp every 5 minutes  
 - schedule: rate(5 minutes)  
 timeout: 20

When we deploy our newly configured Lambda functions, we want to set the stage of our project to better differentiate where we are in development vs. production at any given time. We will use $ sls deploy --stage $STAGE as the command to set the current stage of our project when we deploy our *backend functions*. To help us tell our application what stage we are working on, we will declare a stage in our CloudFormation template as shown above using: stage: ${opt:stage, self:provider.stage}. This mechanism will tell the Serverless Framework and CloudFormation the following:

1. When deploying, CloudFormation should begin to look in opt:stage first, which is the argument that will be passed into the terminal when deploying a service. If the condition is not met, then the same statement tells the Serverless Framework to deploy the new service to self:provider.stage instead, as the stage declared in the provider block.
2. The next line tells us that the stage that we use to deplopy our new services will determine the name of our table: tableName: ${self:custom.stage}-invoices. The best practice is to create separate tables dynamically when deploying services to a new environment. To create a clear separation of concerns between the resources that we use for each of our environments, we need to deploy one table each to our dev and prod stages. When this deploys successfully we will have two tables in our AWS account:
   * dev-invoices
   * prod-invoices

The remaining lines about table Throughput above are commented out, but left in place for you to detemine how to statically provision your Read and Write capacity and Throughput with AWS. In practice, you will need your *production* environment to work with a higher capacity than your *development* environment. In our case we are *autoscaling* our resources on AWS so that our application can grow flexibly with the demand from our users in both prod and dev.

With so much configuration to accomplish, I am amazed that you are still following along. If you think back to the discussion we had about **CI/CD** now you understand why an automated form of *Continuous Delivery or Deployment* is ideal. If you make small changes that can be tested and deployed automatically, you only have to accomplish this setup and configuration once, and from then on, your code will update on its own and deploy to your users autonomously with every new and successful change that you make.

Moving forward, we need to add a few permissions that will allow your Lambda functions to access your DynamoDB tables that you deploy with *Infrastructure As Code*. In the provider block of your serverless.yml file, go ahead and add the iamRoleStatements section that you see below.

provider:  
 name: aws  
 runtime: nodejs10.x  
 stage: dev  
 region: us-east-1  
  
 # Environment variables made available through process.env  
  
 environment:  
 tableName: ${self:custom.tableName}  
  
 # These statements define the acceptable permission policy for our lambda functions  
 # In this case Lambda functions are give permission to access Dynamo  
 iamRoleStatements:  
 - Effect: Allow  
 Action:  
 - dynamodb:DescribeTable  
 - dynamodb:Query  
 - dynamodb:Scan  
 - dynamodb:GetItem  
 - dynamodb:PutItem  
 - dynamodb:UpdateItem  
 - dynamodb:DeleteItem  
 # Need to restrict IM Role to the specific table and stage  
 Resource:  
 -"Fn::GetAtt": [ GeneralLedgerTable, Arn]

To connect to our database we will need to expose a few arguments through the process.env variables that Node.js provides to us through our terminal. We are using the environment block above to tell the Serverless Framework that we will use our environment variables with our Lambda functions. Our tableName will be exposed to us as a process.env reference.

In the final block above we are using the iamRoleStatments to restrict our Lambda functions to specifically work with our GeneralLedgerTable only. Our Lambda functions for this *serverless + microservice* can only access the GeneralLedgerTable with the explicit permissions listed in the Action block shown.

## User Registration & Authentication with AWS Cognito

Now that we have created our first Lambda function, called CreateInvoice.js, that will PUT an invoice entry into our GeneralLedger table in DynamoDB; we will also need a way to register and authenticate the users of our software. Since we are dealing with money, and the payment of invoices using *credit cards*, we will need a mechanism that will allow us to securely register users while safeguarding their private and **Personally Identifiable Information (PII)**, and financial data. Much like Amazon IAM, or **Identity and Access Management**, we will use AWS Cognito in this section to add a layer of **security to our application** that will allow the user to *create, read, update, and delete* invoice and transactionary data within our new [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api) application.

Using [AWS Cognito](https://aws.amazon.com/cognito), we will be able to easily implement user registration, authentication, authorization, and management for our application. [AWS Cognito](https://aws.amazon.com/cognito) is flexible enough to let us implement SSO using Federated Identities with third party Identity Providers (IdP) like Facebook, LinkedIn, or Twitter also. Our user-pool on Amazon Cognito will manage and handle the load of responses that include the authorization tokens returned from each of these social media sign-in *federations* and **SAML IdPs**.

User Pools and Identity Pools are the two primary components that you need to consider when implementing Amazon Cognito. You have a choice of implementing these components together or independently of each other. There are a few [**Common Amazon Cognito Scenarios**](https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-scenarios.html) that you can read about, but in our case we will be concerned with authenticating the users of our application to authorize them to access and use other AWS services that extend our platform.

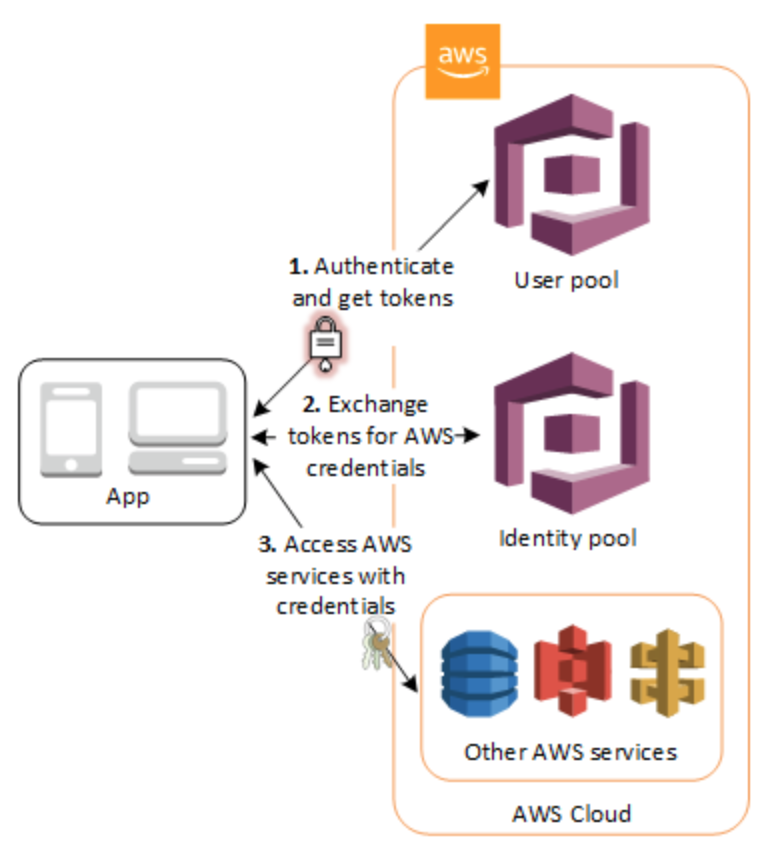
### Implementing both Cognito User Pool and Identity Pool services together

The typical user of the [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api) application will require access to other AWS services like DynamoDB, S3, API Gateway, and more that we may declare throughout the implementation of this application after they register and log into our software. We must properly deploy a strategy that will result in the security of our application, and the effective use of our tool set which will let our user both register and access the services on our platform that are provided to us by AWS.

Amazon Cognito is both **PCI DSS** and **HIPAA** compliant and is ready to deploy to *production*. The idea is to create a *directory of users* with your user-pool that will allow users to sign in to our application. AWS-Amplify is the Amazon SDK that lets us access the user profiles that are created for each of our users whether they sign up with our implementation of Cognito, or using a *federated identity* through a thrid-party **IdP** that we enable on our application.

On the other hand, the identity-pool that we create, will give our users temporary access credentials, much like our IAM service roles provide, so that our user can obtain access to the AWS services and resources like DynamoDB and S3, that we are using to extend the features and functionality of our application. Using both a user-pool and an identity-pool together in Amazon Cognito, we will implement a three step process that resembles the following image:

**Users and their Identities, Together forever**



1. First, when the user authenticates themselves against our user-pool, Cognito will return a token that the application will use to validate the user’s authentication status as they navigate the app.
2. After the user authenticates against the user-pool, the application will map the user’s token to the AWS credentials provided by the Cognito identity-pool.
3. The AWS credentials provided to our user by our application’s identity-pool are now available to use as credentials for services and resources like DynamoDB and S3.

### Creating a Cognito User-Pool

*Infrastructure As Code* is a great way to implement Cognito and the tool set you will use on your application to deploy registration, authentication, and authorization of a user. You can complete the implementation of a user-pool on the AWS Console, but we will focus on completing what we need programatically so that we can commit all of our changes to source so that our **CI/CD** pipeline can deploy our changes automatically.

We will need to create a new file and $ touch ~/services/invoice-log-api/resources/CognitoUserPool.yml so that we can create a CloudFormation template with the ServerlessFramework to tell AWS how to configure our user-pool. Copy and Paste the code below to configure our user *authentication* service in Cognito correctly:

Resources:  
 CognitoUserPool:  
 Type: AWS::Cognito::UserPool  
 Properties:  
 # Dynamically create a name using the correct stage  
 # Make sure you differentiate from other apps on AWS!  
 UserPoolName: ${self:custom.stage}-InvoiceUserPool  
 # Set the alias using email address  
 UsernameAttributes:  
 - email  
 AutoVerificationAttributes:  
 - email  
  
 CognitoUserPoolClient:  
 Type: AWS::Cognito::UserPoolClient  
 Properties:  
 # Dynamically create the appClient name using the correct stage  
 # Make sure you differentiate from other apps on AWS!  
 ClientName: ${self:custom.stage}-PayMeUserPoolClient  
 UserPoolId:  
 Ref: CognitoUserPool  
 ExplicitAuthFlows:  
 - ADMIN\_NO\_SRP\_AUTH  
 GenerateSecret: false  
  
# Output the ID of the user-pool  
Outputs:  
 UserPoolId:  
 Value:  
 Ref: CognitoUserPool  
  
 UserPoolClientId:  
 Value:  
 Ref: CognitoUserPoolClient

At this point it should start to look like more of the same as it pertains to your serverless.yml file. What you should notice after we complete this implementation of our application’s user-pool as *Infrastructure As Code*, is that we are using the ~/services/invoice-log-api/resources directory as a tool to make our *Operations* or *Network Infrastructure* code more **modular**. In the next section we will show you how we reference the user-pool module as a serverless.yml resource, for now though, let’s take a look at the way we implemented our user-pool:

1. The UserPoolName property tells us that the stage that we use to deploy our service will determine the name of our user-pool with the statement: UserPoolName: ${self:custom.stage}-InvoiceUserPool. The best practice is to create a separate user-pool dynamically when deploying services to a new environment. To create a clear separation of concerns between the resources that we use for each of our environments, we need to deploy one user-pool each to our dev and prod stages. When this deploys successfully we will have the following resources in our AWS account:
   * dev-InvoiceUserPool
   * prod-InvoiceUserPool
2. We want our user to log into our application with their *email address* as their username. With our CloudFormation template we tell Cognito and our new \*-InvoiceUserPool, to set the UsernameAttributes to email as shown above.
3. Finally, using the Outputs block shown, we tell CloudFormation to return the UserPoolId and the UserPoolClientId that we generate dynamically so that we can reference it later.

#### Adding your InvoiceUserPool as a serverless.yml Resource

Just like we reference our modular DynamoDB table in our ~/services/invoice-log-api/resources directory from our serverless.yml file, we will have to do the same thing for our user-pool by replacing the resources block in our CloudFormation template with the information below:

# Keep resources modular and create each with separate CloudFormation templates  
resources:  
 # DynamoDB Service  
 - ${file(resources/GeneralLedgerTable.yml)}  
  
 # Cognito User-Pool Service  
 - ${file(resources/CognitoUserPool.yml)}

### Creating a Cognito Identity-Pool

Earlier we discussed using a Cognito identity-pool to give our users temporary access credentials, much like our IAM service roles provide, so that our user can obtain access to the AWS services and resources like DynamoDB and S3, that we are using to extend the features and functionality of our application. We need to complete this with *Infrastructure As Code* so that our Identity-Pool knows that it should use the User-Pool above to authenticate our applications’s *users*.

We will need to create a new file and $ touch ~/services/invoice-log-api/resources/CognitoIdentityPool.yml so that we can create a CloudFormation template with the ServerlessFramework to tell AWS how to configure our identity-pool. Copy and Paste the code below to configure our identity-pool correctly:

Resources:  
 # This is the fed IdP that will auth with our user-pool  
 CognitoIdentityPool:  
 Type: AWS::Cognito::IdentityPool  
 Properties:  
 # Dynamically create a name using the correct stage  
 # Make sure you differentiate from other apps on AWS!  
 IdentityPoolName: ${self:custom.stage}-InvoiceIdentityPool  
  
 # This will prevent unauthenticated access to platform  
 AllowUnauthenticatedIdentities: false  
  
 # Configure User Pool  
 CognitoIdentityProviders:  
 - ClientId:  
 Ref: CognitoUserPoolClient  
 ProviderName:  
 Fn::GetAtt: [ "CognitoUserPool", "ProviderName" ]  
  
 # IAM Role for user-pool  
 CognitoIdentityPoolRoles:  
 Type: AWS::Cognito::IdentityPoolRoleAttachment  
 Properties:  
 IdentityPoolId:  
 Ref: CognitoIdentityPool  
 Roles:  
 authenticated:  
 Fn::GetAtt: [CognitoAuthRole, Arn]  
  
 # IAM role for authentication of a user  
 CognitoAuthRole:  
 Type: AWS::IAM::Role  
 Properties:  
 Path: /  
 AssumeRolePolicyDocument:  
 Version: '2012-10-17'  
 Statement:  
 - Effect: 'Allow'  
 Principal:  
 Federated: 'cognito-identity.amazonaws.com'  
 Action:  
 - 'sts.AssumeRoleWithWebIdentity'  
 Condition:  
 StringEquals:  
 'cognito-identity.amazonaws.com:aud':  
 Ref: CognitoIdentityPool  
 'ForAnyValue:StringLike':  
 'cognito-identity.amazonaws.com:amr': authenticated  
 Policies:  
 - PolicyName: 'CognitoAuthorizedPolicy'  
 PolicyDocument:  
 Version: '2012-10-17'  
 Statement:  
 - Effect: 'Allow'  
 Action:  
 - 'mobileanalytics:PutEvents'  
 - 'cognito-sync:\*'  
 - 'cognito-identity:\*'  
 Resource: '\*'  
  
 # Users have permission to invoke API  
 - Effect: 'Allow'  
 Action:  
 - 'execute-api:Invoke'  
 Resource:  
 Fn::Join:  
 - ''  
 -  
 - 'arn:aws:execute-api:'  
 - Ref: AWS::Region  
 - ':'  
 - Ref: AWS::AccountId  
 - ':'  
 - Ref: ApiGatewayRestApi  
 - '/\*'  
  
 # Users have permission to upload to their folder in s3  
 - Effect: 'Allow'  
 Action:  
 - 's3:\*'  
 Resource:  
 - Fn::Join:  
 - ''  
 -  
 - Fn::GetAtt: [AttachmentsBucket, Arn]  
 - '/PrivateDirectory/'  
 - '$'  
 - '{cognito-identity.amazonaws.com:sub}/\*'  
  
# Output the ID of the identity-pool  
Outputs:  
 IdentityPoolId:  
 Value:  
 Ref: CognitoIdentityPool

Again, this is pretty much the same thing we are doing in the *Infrastructure As Code* implementation for our Cognito user-pool. Our CloudFormation templates make this look more complex than it is, and if you just take some time to read through the property and resource declarations in the file you should be able to make sense of this while reconciling the information deployed here with the information on your AWS Console and what it shows you in Cognito.

1. Just like we have done for every other service on AWS that we have implemented with *Infrastructure As Code*, we name our identity-pool depending on the stage that we deploy our service to from the terminal. We name our identity-pool using the statement: IdentityPoolName: ${self:custom.stage}-InvoiceIdentityPool.
2. We have to explicitly declare that only users who log into our applicagtion with Cognito are authorized to access the root directory of our application with the next line of *IAC* shown: AllowUnauthenticatedIdentities: false
3. You will not believe this, but we also have to tell AWS that our IdP will authorize the users in our directory to use the AWS services we have deployed. The statement below references the CognitoUserPoolClient we defined in our user-pool in the previous section:

# Configure User Pool  
 CognitoIdentityProviders:  
 - ClientId:  
 Ref: CognitoUserPoolClient

1. We have to consider that the users we authenticate will need our application to attach the IAM roles to the identities in our user-pool directory. The implementation above is formatted according to CloudFormation *Best Practices*
2. The ServerlessFramework creates an ApiGatewayRestApi **Ref** everytime we implement a serverless endpoint in our serverless.yml file. The - Ref: ApiGatewayRestApi statement just refers back to the resource deployed to API Gateway by CloudFormation.

The final result of the implementation above is the IdentityPoolId that is returned to our application by the Outputs block. This is what our application uses to make use of the services we have deployed on AWS to support our application in the *wild* later on.

#### Adding your InvoiceIdentityPool as a serverless.yml Resource

Just like we reference our modular DynamoDB table in our ~/services/invoice-log-api/resources directory from our serverless.yml file (and just like our implementation of the user-pool), we will have to do the same thing for our identity-pool by replacing the resources block in our CloudFormation template with the information below:

# Keep resources modular and create each with separate CloudFormation templates  
resources:  
 # DynamoDB Service  
 - ${file(resources/GeneralLedgerTable.yml)}  
  
 # Cognito User-Pool and Identity-Pool Services  
 - ${file(resources/CognitoUserPool.yml)}  
 - ${file(resources/CognitoIdentityPool.yml)}

## Implement the remaining Lambda functions

### Implement getInvoice.js

We need a mechanism to get an individual invoice that we need to display to a user of our application. This funcion will need to GET an invoice for those who create them and those who receive them from another user. We need to make sure we implement a mechanism that will only let a user see the invoices that pertain to their specific account.

// Step1: Discuss in tutorial why /lib and response-lib  
// exist and why they are used. (See create)  
import { success, failure } from "./libs/responseLib";  
  
// Discuss in tutorial the need for a Dynamo table  
// and a dynamo table implemented for each microservice  
// Implement service below  
import \* as dynamoLib from "./libs/dynamoLib";  
  
export async function main(event, context) {  
 // Request body is passed in as a JSON string in event.body!  
 const data = JSON.parse(event.body);  
  
 // Same as in createInvoice we need to declare the object  
 // that we want to get from the database this time  
 const params = {  
 TableName: process.env.tableName,  
 /\* 'Key': this is what will define our partition & sort  
 \* key for the item that we have to retrieve  
 \*  
 \* - 'userId': is the IdP identity of the authenticated user  
 \* - 'invoiceId': is the path parameter that we must include  
 \* in the request to this service  
 \*/  
  
 Key: {  
 userId: event.requestContext.identity.cognitoIdentityId,  
 invoiceId: event.pathParameters.id  
 }  
 };  
  
 try {  
 const result = await dynamoLib.call("get", params);  
 if (result.Item) {  
 //return the item that we retrieved  
 return success(result.Item);  
 } else {  
 return failure({  
 status: false,  
 error: "Item was not found in the DB."  
 });  
 }  
 } catch ( err ) {  
 return failure({  
 status: false  
 });  
 }  
}

### Implement deleteInvoice.js

We need a mechanism to delete an individual invoice that we have saved in our application’s database under a specific user. This funcion will need to DELETE an invoice as triggered by the user who created the invoice.

import { success, failure } from "./libs/responseLib";  
  
// Discuss in tutorial the need for a Dynamo table  
// and a dynamo table implemented for each microservice  
// Implement service below  
import \* as dynamoLib from "./libs/dynamoLib";  
  
export async function main(event, context) {  
 const params = {  
 TableName: process.env.tableName,  
 /\* 'Key': this is what will define our partition & sort  
 \* key for the item that we have to retrieve  
 \*  
 \* - 'userId': is the IdP identity of the authenticated user  
 \* - 'invoiceId': is the path parameter that we must include  
 \* in the request to this service  
 \*/  
  
 Key: {  
 "userId": event.requestContext.identity.cognitoIdentityId,  
 "invoiceId": event.pathParameters.id  
 }  
 };  
  
 try {  
 const result = await dynamoLib.call("delete", params);  
 return success({  
 status: true  
 });  
 } catch ( err ) {  
 return failure({  
 status: false  
 });  
 }  
}

### Implement updateInvoice.js

We need a mechanism to update an individual invoice that we have saved in our application’s database under a specific user. This funcion will need to UPDATE an invoice as triggered by the user who created the invoice.

import { success, failure } from "./libs/responseLib";  
  
// Discuss in tutorial the need for a Dynamo table  
// and a dynamo table implemented for each microservice  
// Implement service below  
import \* as dynamoLib from "./libs/dynamoLib";  
  
export async function main(event, context) {  
 const data = JSON.parse(event.body);  
  
 const params = {  
 TableName: process.env.tableName,  
 /\* 'Key': this is what will define our partition & sort  
 \* key for the item that we have to retrieve  
 \*  
 \* - 'userId': is the IdP identity of the authenticated user  
 \* - 'invoiceId': is the path parameter that we must include  
 \* in the request to this service  
 \*/  
  
 Key: {  
 // Need a tutorial setting up Cognito!!!!  
 "userId": event.requestContext.identity.cognitoIdentityId,  
 "invoiceId": event.pathParameters.id  
 },  
 /\*  
 \* 'UpdateExpression' defines the attributes to update  
 \*  
 \* 'ExpressionAttributeValues' defined the value that  
 \* that we need in UpdateExpression  
 \*  
 \*/  
 UpdateExpression: "SET content = :content, :attachment",  
 ExpressionAttributeValues: {  
 ":attachment": data.attachment || null,  
 ":content": data.content || null  
 },  
 /\*  
 \* 'ReturnValues' specifies if and how to return our Item's  
 \* attributes, where ALL\_NEW returns all attributes of the  
 \* item after the update.  
 \*  
 \* Inspect 'result' to verify the values of the different  
 \* applications settings that we configure  
 \*  
 \*/  
 ReturnValues: "ALL\_NEW"  
 };  
  
 try {  
 const result = await dynamoLib.call("update", params);  
 return success({  
 status: true  
 });  
 } catch ( err ) {  
 return failure({  
 status: false  
 });  
 }  
}

### Implement listInvoices.js

We need a mechanism to list all of the invoices that we have saved in our application’s database for a specific user. This funcion will need to GET all invoices created by a user.

import { success, failure } from "./libs/responseLib";  
  
// Discuss in tutorial the need for a Dynamo table  
// and a dynamo table implemented for each microservice  
// Implement service below  
import \* as dynamoLib from "./libs/dynamoLib";  
  
export async function main(event, context) {  
  
 const params = {  
 TableName: process.env.tableName,  
 /\*  
 \* KeyConditionExpression will define the condition for the  
 \* query.  
 \* - 'userId = :userId' will only return the Items tht match  
 \* the 'userId' PartitionKey.  
 \*  
 \* ExpressionAttributeValues will define the value in the  
 \* condition.  
 \* - ':userId': will define a 'userId' that maps to the  
 \* Identity Pool Cognito Identity Id of the  
 \* authenticated user!!!  
 \*  
 \*/  
  
 KeyConditionExpression: "userId = :userId",  
 ExpressionAttributeValues: {  
 ":userId": event.requestContext.identity.cognitoIdentityId  
 }  
 };  
  
 try {  
 const result = await dynamoLib.call("query", params);  
 // return the matching list of invoices in response.body!  
 return success(result.Items);  
 } catch ( err ) {  
 return failure({  
 status: false  
 });  
 }  
}

### You completed the implementation of your serverless backend application and logic. Good Luck!

## Part 6: Build a React.js Frontend for your PayPal clone. Take home the Cash with the PayMyInvoice App!

* [Part 6: Build a React.js Frontend for your PayPal clone](https://github.com/lopezdp/TechnicalArticles/blob/master/BuildAReactJsFrontendforYourPayPalClone.md) - *Not Published.*