

The Definitive Guide to AWS Application Integration

With Amazon SQS, SNS, SWF and
Step Functions

Jyothi Prasad Buddha
Reshma Beesetty



Apress®

The Definitive Guide to AWS Application Integration

With Amazon SQS, SNS, SWF
and Step Functions

Jyothi Prasad Buddha
Reshma Beesetty

Apress®

The Definitive Guide to AWS Application Integration

Jyothi Prasad Buddha
Seattle, WA, USA

Reshma Beesetty
Seattle, WA, USA

ISBN-13 (pbk): 978-1-4842-5400-4
<https://doi.org/10.1007/978-1-4842-5401-1>

ISBN-13 (electronic): 978-1-4842-5401-1

Copyright © 2019 by Jyothi Prasad Buddha and Reshma Beesetty

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: Celestin Suresh John

Development Editor: James Markham

Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5400-4. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

*To our parents,
Vijaya Parvati and Venkata Ramana Buddha,
Lakshmi Tulasi and Venkata Ramana Beesetty,
whose love and support shaped us into
who we are today*

Table of Contents

| | |
|--|-------------|
| About the Authors..... | xi |
| About the Technical Reviewer | xiii |
| Acknowledgments | xv |
| Introduction | xvii |
| | |
| Chapter 1: Getting Started | 1 |
| Available Services..... | 2 |
| Basic Concepts | 5 |
| IAM | 6 |
| Simple Storage Service | 10 |
| DynamoDB | 18 |
| Lambda..... | 35 |
| Sample Application | 49 |
| Understanding the Project..... | 52 |
| Vue Components..... | 54 |
| Summary..... | 57 |
| | |
| Chapter 2: Simple Queue Service | 59 |
| Big Picture..... | 64 |
| Creating a Queue | 66 |
| Console Method..... | 66 |
| CLI Method | 70 |
| SDK Method..... | 74 |

TABLE OF CONTENTS

| | |
|--|------------|
| Send Messages to Queues..... | 77 |
| Console Method..... | 78 |
| CLI Method | 80 |
| SDK Method..... | 82 |
| Pull Message from a Queue | 84 |
| Console Method..... | 84 |
| CLI Method | 86 |
| SDK Method..... | 88 |
| Long Polling..... | 92 |
| Life Cycle of a Message | 93 |
| Consumption Approaches | 96 |
| Lambda SQS Worker..... | 98 |
| EC2 NodeJS SQS Worker | 112 |
| FIFO (First-In-First-Out) Queues..... | 126 |
| Creating FIFO Queues | 127 |
| Sending Messages | 128 |
| Receive Messages..... | 130 |
| Retry Failed Receive Requests | 132 |
| Dead Letter Queues | 132 |
| Handle Messages in Dead Letter Queues..... | 137 |
| Summary..... | 137 |
| Chapter 3: Simple Notification Service..... | 139 |
| Difference from SQS | 140 |
| Setting Up | 141 |
| Creating Topics | 142 |
| Create Subscription | 146 |
| Publishing Messages | 150 |
| API Method | 153 |

TABLE OF CONTENTS

| | |
|---|------------|
| Message Receivers | 158 |
| Lambda Receiver..... | 158 |
| SQS Receiver..... | 162 |
| Message Filtering | 163 |
| Filter Expressions | 168 |
| Push Notifications | 173 |
| Firebase Steps..... | 174 |
| Initializing Firebase in web-app | 176 |
| SNS Setup | 179 |
| Client Endpoint | 182 |
| Testing It Out | 183 |
| Miscellaneous Topics | 184 |
| Access Policy..... | 185 |
| Summary..... | 187 |
| Chapter 4: Simple Workflow Service | 189 |
| Architecture of SWF Applications..... | 190 |
| Creating Types..... | 193 |
| Register Domain | 193 |
| Register Workflow Type | 199 |
| Register Activity Types..... | 203 |
| Workflow Execution | 208 |
| Implementing Actors | 211 |
| Implementing Starter..... | 212 |
| Implementing Decider | 217 |
| Implementing Activity Worker..... | 233 |
| Serverless Workflow | 237 |
| Serverless Activity Workers | 238 |
| Serverless Deciders..... | 242 |

TABLE OF CONTENTS

| | |
|--|------------|
| Advanced Topics | 248 |
| Timers..... | 248 |
| Recording Heartbeats..... | 253 |
| Miscellaneous Topics | 255 |
| Frameworks..... | 256 |
| Pricing | 257 |
| Limits..... | 259 |
| Summary..... | 260 |
| Chapter 5: Step Functions | 263 |
| Key Concepts | 264 |
| State Machine | 264 |
| Tasks | 265 |
| Activity..... | 265 |
| Basic Operations..... | 266 |
| Creating State Machine | 266 |
| Executing the State Machine..... | 272 |
| API Method | 273 |
| State Machine Definition..... | 277 |
| Top-Level Fields..... | 277 |
| States | 278 |
| Activities | 309 |
| Creating and Using Activity..... | 309 |
| Activity Worker Implementation | 311 |
| Other Patterns..... | 317 |
| Long-Running Tasks | 317 |
| Child Step Functions..... | 323 |
| Creating Loops..... | 324 |

TABLE OF CONTENTS

| | |
|------------------------------|------------|
| Monitoring..... | 329 |
| Updating Order Workflow..... | 333 |
| Step Functions vs. SWF | 338 |
| Pricing..... | 340 |
| Summary..... | 341 |
| Index..... | 343 |

About the Authors



Jyothi Prasad Buddha has 12+ years of experience in software development along with building tools for optimizing testing. He has worked on many time-critical projects for IBM, Oracle, and Amazon, by inventing new time-saving approaches and tools consistently. At Amazon, he developed two products named LiveCode and HireOn quickly using Amazon Web Services (AWS) which were experiments from Amazon and later became full-fledged products embedded into Amazon's recruiting workflow. He is also a Pluralsight author and an open source contributor and was a Java User Group Hyderabad lead. He is currently working for Amazon, based out of Seattle, WA.



Reshma Beesetty is currently working as a software developer for Tata Consultancy Services. She is also a published author with experience in extensive research-oriented content creation. She has produced and managed content for several startups by designing and maintaining editorial calendars that were published across various web platforms. Her work as an article writer and report creator has been published across state media of Andhra Pradesh, Karnataka, and Tamil Nadu in India.

About the Technical Reviewer



Anindita Basak is a cloud architect and DevOps engineer. With over a decade of experience, she helps enterprises to enable their digital transformation journey empowered with Multicloud, DevOps, Advanced Analytics, and AI. She co-authored *Stream Analytics with Microsoft Azure* and *Hands-On Machine Learning with Azure* and was a technical reviewer of seven books on Azure along with two video courses on Azure Data Analytics. She also worked extensively on AWS Infra, DevOps, and Analytics.

She'd like to thank her parents and brother for their unbounded support and the Apress-Springer team!

Acknowledgments

Writing a book, as opposed to what it may seem, is never a one-man job. We would like to use this space for thanking each and every one who helped materialize the book you are holding in your hands right now, from random design thoughts in our heads to actual ink on the paper or e-ink on your Kindle.

We first want to extend our sincere thanks to Samrat Kakustam who spent countless hours on discussion of crucial design choices for the book throughout. His inputs in our brainstorming sessions proved to be quintessential for achieving the desired outcome in each and every chapter. He has provided us with valuable comments for every chapter on how best to demonstrate usefulness of the services.

This wouldn't have been possible without the most ambitious and capable team from Apress. Special thanks to Aditee Mirashi who has been constantly pushing us so that we wouldn't miss the deadline. We would also want to thank our technical reviewer, Anindita Basak, and our development editor, James Markham, for the eagle's eye reviews which helped us make the outcome as refined as humanly possible.

And finally, yet the most crucial, there with us from the literal beginning were our parents. We would like to thank them for helping us to stand on our own feet. It feels incomplete without acknowledging their efforts that are not directly visible ever. Thanks to you guys.

Introduction

If we look beyond the basic AWS services everyone starts out with, we realize that there are many other useful services. This book helps the reader look beyond the basic AWS services and opens the doors for building interesting applications with many components that communicate and coordinate with each other by demonstrating how to send messages, orchestrate tasks, and create state machines.

We begin at the very basics. Even readers who are not familiar with AWS will be able to follow through effortlessly. Chapter 1 covers all the prerequisites such as Elastic Cloud Compute (EC2), Lambda, Simple Storage Service (S3), DynamoDB, CloudWatch, and Identity and Access Management (IAM). We then proceed to the simplest mechanism for service coordination, sending messages using queues and topics in the next two chapters.

Chapters 2 and 3 cover Simple Queue Service (SQS) and Simple Notification Service (SNS). In Chapter 4, we learn how to build workflows for simple orchestration of tasks. We learn about Simple Workflow (SWF) Service in this chapter along with creating different types of deciders and activity workers. In Chapter 5, we raise the bar by showing why we need state machines and how to build them with the help of Step Functions (SFn). Every chapter teaches how to use these services by means of console, SDK, as well as command line interface (CLI).

INTRODUCTION

By the end of this book, you will have achieved a complete understanding of basic concepts such as how to build individual components along with advanced concepts such as integrating distributed components of one or more applications. Most of the code examples are in JavaScript, and a few examples of Java and Python are also provided in the book, and more examples will be provided through the GitHub repository. The book is heavy with practical hands-on exercises of all the topics, just like we intended it to be.

CHAPTER 1

Getting Started

With the advent of cloud platforms such as Amazon Web Services (AWS), developing applications that solve interesting problems and scale them to fit the usage is easy nowadays. However, any new developer who wants to start with AWS finds out soon enough that it's wise to first spend time understanding how the solutions offered by AWS map to the business requirements. Since AWS, platform-wise, has more than 100 services as of writing this book, even the experienced developers often get overwhelmed by the sheer number of services available in the offering. More often than not, developers just end up using a few frequently used services to achieve their means or to develop applications.

The obvious EC2, DynamoDB, S3, and Lambda are low-hanging fruits and can help us develop a plethora of applications. These serve most needs and can solve many use cases without using other services. Even the first ever application I have developed for Amazon only uses services as mentioned earlier along with CloudWatch and Elastic Container Service (ECS).

There are many tutorials available over the Internet that show how to stitch together multiple Lambda functions to implement long-running computations, run entire web sites serverless, and many more. If not for sounding clichéd, I want to repeat the phrase “just because we can does not mean we should.” We can certainly use a hammer for bringing down a tree, but an ax would work better. They work for quick prototyping and developing a minimum viable product or MVP. However, these tricks

end up becoming a technical debt as your project grows. Maintenance becomes a burden, and debugging production issues often turns into a headache for on call.

Many use cases are better handled by using Application Integration services of AWS. These are a group of products developed for letting applications interact with each other in a cohesive and fault-tolerant way. These services help us develop distributed applications that are reliable and can scale well both in terms of performance overhead and being soft on development and maintenance. We will learn all of these services to avoid incurring tech debt and build modern apps, but let us not get ahead of ourselves yet. We have a chapter dedicated to cover each integration service or product.

However, before we get there, we need to understand a few fundamentals. As you know that the integration products are for blending applications together, we first need to learn how to build these independent applications using other services of AWS. Once we grasp that, we can go on about our business of integrating them. So let us discover a few essential services of AWS before we dive deep into integration products.

Available Services

As of 2019 Q1, there are 138 services/products in AWS, and the number doesn't seem to reduce in the near future. We can never cover all of them in one book to avoid the risk of boring you to death. I will introduce a few crucial sections of AWS. Here is a partial look at the AWS Management Console in Figure 1-1 while writing this book. The integration products, the subject matter for this book, are surrounded by a rectangle. This should throw a light on the vast scale of services AWS offers via numerous service categories, even though not all of them are in the scope of our book.

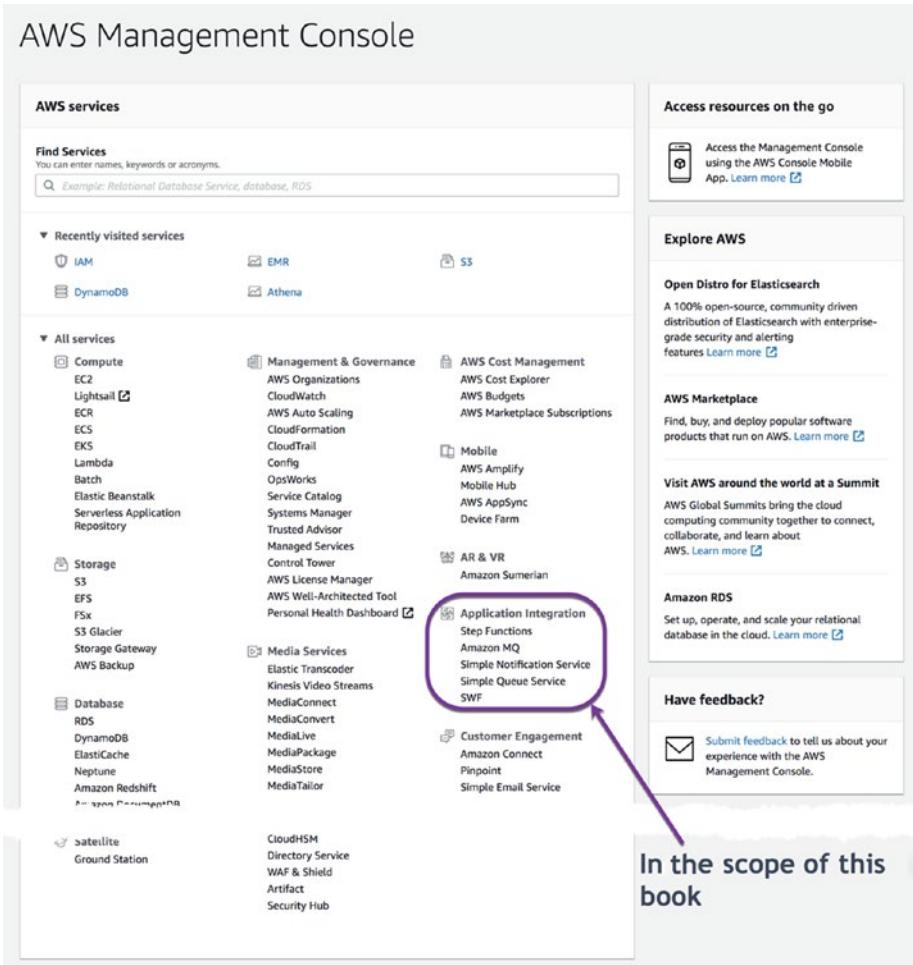


Figure 1-1. AWS Management Console Lists All the Services

As you can observe in Figure 1-1, the AWS console divides all the services into sections. Each section deals with specific requirements we encounter in developing a cloud or for that matter on-premise applications as well. The Compute section of AWS has several services that enable us to create servers/virtual machines onto which we can deploy our applications. Any non-client-side code that needs to be executed has to be deployed on a server. We can deploy it on a virtual machine that can be

created using EC2 service or use a serverless Lambda function. Lambda functions allow you to run scalable code with no need for you to provision a server. The Elastic Container Service is for running containerized applications using Docker. The best part of AWS is that you can create a distributed application that runs on a multitude of these services that can communicate with each other easily.

The Storage section of AWS tackles the problem of handling files necessary for our applications to run. S3 is one such simple service that can store or retrieve a file without creating and managing a server or any hardware. Be it image files for a service like Instagram or a large amount of data for big data processing, S3 can store them all. The Database portion of AWS is for serving live data that needs to be queried and retrieved. The following is the list of some of the database services and their corresponding use case scenarios:

- RDS, Relational Database Service, allows you to use a traditional RDBMS such as Oracle, MySQL, or even Amazon's own Aurora.
- DynamoDB is a NoSQL database that guarantees lightening quick querying of data.
- Redshift is a fast and scalable data warehouse that makes it easy and cost-effective to analyze petabytes of data.
- Neptune is a graph database service aimed at facilitating the applications to work with connected datasets and navigating relationships.

Analytics is a group of services that simplify adhoc analysis of the data stored in a data warehouse or S3 or many other non-AWS storages. For example, Amazon Athena allows users to run ad hoc queries on S3 data. QuickSight allows users to create and publish interactive dashboards including ML Insights.

The services under Cost Management allow you to budget and track your spending in AWS. CloudWatch, CloudTrail, and few other services that facilitate development and debugging fall under the Management and Governance section, while services like Identity and Access Management (IAM) and Cognito fall under the Security, Identity, and Compliance section of the management console.

Basic Concepts

Let us now learn a few basic concepts of the services we have to use in this book. Remember that teaching all the technicalities of the services is not the focus of this book; hence, we go over them pretty quickly. We first need to create a free AWS account for you; and before we do that, however, I should warn you that not everything in AWS is free.

There are few services like DynamoDB which have a free tier of 25 GB or Lambda functions that allow 1 million free invocations every month forever. However, some services are only free for the first 12 months after you sign up, which means they have a free tier like EC2 which allows you to run a small capacity server (t2.micro instance) for 750 hours per month. However, this free tier is only available for 1 year. There are services like LightSail that give a short-term free trial for one or two months. You can achieve a lot with these generous free trials, but before you work with any service, read its pricing to know what to expect.



Caution There is no free tier for some AWS services. Even for the services that do have a free tier, you need to pay once you cross the free limits.

Once we got the warning out of the way, you can create a free AWS account by following the set of steps listed below:

1. Open the web page <https://aws.amazon.com/free> and click Create a Free Account.
2. Fill in an email address, password, and account name and click Continue.
3. Choose the account type as personal and fill in a phone number, full name, and address.
4. It then asks you to provide credit/debit card details.
5. In the next page, you need to confirm your identity using an SMS or voice call.
6. Once the verification is complete, choose a support plan. You can pick the free plan.

Once you finish all the steps outlined, you can sign in to the console by clicking the ***Sign In to the Console*** button on the final confirmation page. Log in with the email address and password you have used while setting up the account to open the ***AWS Management Console*** page. In this page, you can access all the services offered by AWS and use them to your heart's desire. Terms and conditions apply as usual.

IAM

Identity and Access Management or IAM is the service we should first touch upon, because this allows us to create users and roles our applications need. We don't want to let unauthorized users fiddling with our datasets or applications we deploy in AWS. For this, the IAM service allows us to define users and roles that can have access to the services or

resources. We can give access to a user or a role and can control various access-related aspects. Some of the examples of fine-grained access possible through this service go like

- Which users should be able to log in to the console?
- Who can upload data to an S3 bucket?
- Who can create DynamoDB tables or manage capacity for them?

To open IAM, find it in the AWS Management Console or type in the search bar. If you open IAM, you should find the UI easy enough to work with as shown in Figure 1-2.

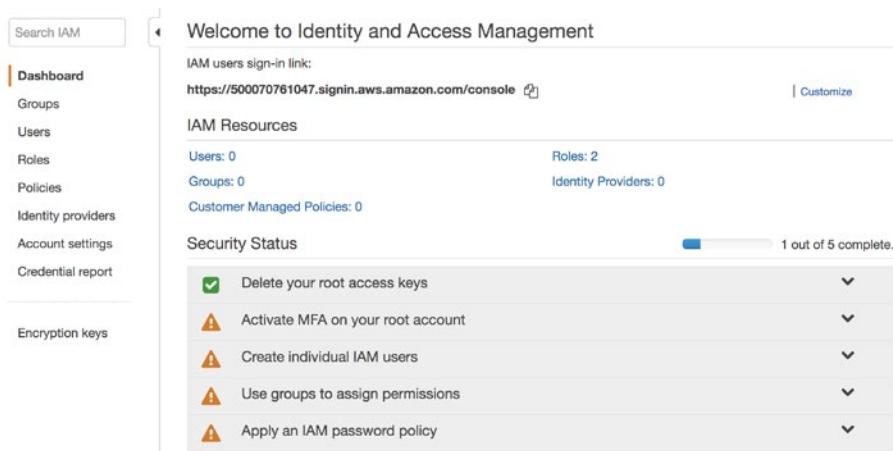


Figure 1-2. IAM Console Home Page

We can use the left menu of the service to switch between different views in IAM. We are in the Dashboard at the moment. You can find a URL which also includes your account id that can be used to log in to the console. If you glance at the Security Status segment of the Dashboard, it doesn't look pretty. It has many warnings displayed on the screen. They are all best practices we have to follow to keep our AWS accounts safe and

secure. MFA or multifactor authentication adds extra layer of protection. Another aspect to keep in mind is that we logged in as a root user at the moment. As a root user, we can perform any operation. So please create another user because only the root user can perform some critical tasks on the AWS account. By using the root user for everyday tasks, we risk losing it or other developers gaining access. I recommend enabling MFA for the root user, but it is optional. If you wish to do so, click the warning “Activate MFA on your root account” in the Dashboard and follow the on-screen instructions. I would strongly recommend you do so, as it is one of the best practices to keep your AWS account secure.

Creating an Admin User

As we have established that we don't want to use the root user for normal operations, let us create a new user with administrative privileges. To create one, navigate to the user segment of IAM by clicking the Users hyperlink in the sidebar. You wouldn't be seeing any users yet. The root user is not visible here. Click Add User, give the name as admin, and choose both access types. Programmatic access allows users to interact with AWS through CLI or programmatically. In other words, you can write a program that can perform allowed actions using the Access Key ID and Secret Key of this user. By selecting AWS Management Console Access, the user can log in to AWS as a root user did. Choose both for our admin user because as I suggested earlier, we should avoid performing day-to-day management tasks using the root user as a best practice. You can set the console password as the auto-generated or custom password.

The next screen asks you to choose the permissions this user needs. You can choose **“Attach existing policies directly”** to use a preconfigured permission policy. A policy is a set of permissions that say

what a user may do. If you click the arrow beside the policy, you can see the JSON syntax of the policy. I'm choosing administrator access policy which has JSON text as in the following:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "*",  
            "Resource": "*"  
        }  
    ]  
}
```

The policy is easy enough to understand. Effect parameter is for stating whether to allow or deny. Action is to list all the operations that can be performed. Resource property is for specifying one or more AWS entities. So you can restrict a user to edit a specific database table by providing the resource here. If you look at the policy for AmazonS3ReadOnlyAccess, you can see that the user is allowed to perform only Get and List on all S3 objects. The statement should look similar to this:

```
{  
    "Effect": "Allow",  
    "Action": ["s3:Get*", "s3>List*"],  
    "Resource": "*"  
}
```

Feel free to browse through other policies to understand the syntax better. The next page deals with tagging, which is a way in AWS to monitor and group expenses, and finally the confirmation page completes the user creation process. If everything is done right, you will be redirected to a success page where you can see and download credentials for the

admin user. Credentials in AWS are a combination of Access Key ID and Secret Access Key. You should copy and store them somewhere safe as you cannot get the Secret Access Key later. You can create a new set of credentials if you lose these.

You may wonder why we have given a password while creating the user and we now have a separate set of credentials. We use the password to open the console from a browser, while credentials are for programmatic access or CLI access. These credentials can be rotated or changed without affecting applications while using them as long as we follow specific guidelines. It is a mechanism of damage control if an unauthorized hacker gets the credentials. I will show how to use these in upcoming sections.

Along with the credentials, the success page also contains a URL for logging in to the console. Clicking it takes us to a login page which we can use to log in as an IAM user, provided that we gave the new user console access. Log in using the link to change into the new user we created and stop being root. As we are logging in through a browser, you now have to use the IAM user name and the console password you have set (or auto-generated), not the credentials. As we have given administrator access while creating this new user, the user should be able to perform most of the tasks.

EXERCISE

Here comes your first exercise. Create a new user named s3_publisher. The user should have access to upload files to S3. Save the credentials.

Simple Storage Service

Even though the Simple Storage Service (S3) is a well-known name, people are not entirely aware of how widely it is used. Big web sites like Netflix, Airbnb, and Thomson Reuters use it. The big brother of e-commerce Amazon.com uses S3 to deliver its static content like product pictures,

store numerous backups, and whatnot. S3 is a simple service. You create buckets in S3, which are nothing but root folders for your content. You can create folders in the buckets and upload files into the folders. You can create as many buckets as 100 buckets per account by default and the limit can be increased if necessary by raising an AWS support ticket.

Creation of buckets doesn't cost you anything. Neither does creation of folders or sub-folders in buckets. What costs you is the files you put or get. Costs depend on how much GB you store, how many times you request for the files, and a few other parameters. You can go to the pricing page of S3 to understand it better. S3 has a 12-month free tier, which means it is free for new accounts as long as you keep the data under 5 GB. So, as long as you finish this book in a year from creating your account, finishing exercises in this book shouldn't cost anything from S3 point of view. I will remind you whenever we do something that involves cost and potential expense, so you are aware. Money matters aside, let us get to the technical aspects of using S3. Once you log in to the console, click S3 listed under Storage. If you feel like you are looking for a needle in a haystack, type S3 in the search box to find it. Upon opening S3, it should take you to a page which lists the buckets you have created so far.

Hosting a Static Web Site on S3

Once you are inside the S3 service, we can create buckets and upload files into them. One practical application that we can achieve quickly is launching a static web site. A static web site doesn't need any server-side rendering as it is purely a combination of HTML, CSS, JavaScript, and image files. So S3 is a great place to host it on. All the files related to a static web site must be in a bucket, so we would need to create one to upload all our assets. Once we upload assets like index.html, the static web site can be opened by the following URL. You should replace the bucket name with the name you gave for your bucket and the region you created the bucket in. Some regions' URLs have a hyphen before **region**, and others need a dot.

bucket-name.s3-website-*region*.amazonaws.com

or

bucket-name.s3-website.*region*.amazonaws.com

Let us take the necessary actions to host a static web site. Clicking the **Create Bucket** button opens up a wizard that asks for a bucket name, region, options, and permissions. The region is a geographic placement for your files. If you think that all your customers are in India, it makes more sense to keep the files closer to them, so the latency is less. You can also set up cross-region replications or cloud front to cut latency for people who live on the other side of the globe, but it is out of scope here as it is a topic worthy of its own book.

Bucket names are global, which means no one else should have the same bucket name as yours in any region. The options page of the wizard deals with whether you need versioning of files, what tags you want to apply for the bucket, encryption, and so on. You can leave the default values for now. The next page is for setting what kind of read or write permissions are allowed on this bucket. In this exercise, we want to host a static web site, so we need the files to be public, so deselect all four checkboxes. You don't need to deselect all four of them, but let us make our lives easy for this exercise's sake. When the time comes for you to launch your next big project for the whole wide world, read about these permissions thoroughly.

Once you complete reviewing the information, finish the creation of the bucket. You should see your bucket in the list of buckets in the S3 buckets page. Open the bucket by clicking it. I created the bucket with the name *jo-dummy*. So it should look like in Figure 1-3.

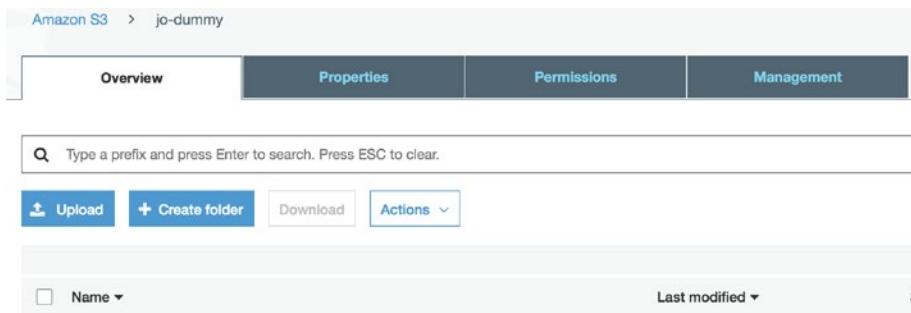


Figure 1-3. Overview Tab of a Bucket

Create a new HTML file named index.html with the following content and upload it to the bucket by clicking the Upload button. In the permissions page of the upload wizard, select “Grant public read access to this object(s)” to allow this file to be accessed publicly:

```
<html lang="en">
  <head>
    <title>Hello S3</title>
  </head>
  <body>
    <h1>Hello from Simple Storage Service</h1>
  </body>
</html>
```

If you try to access the web site as soon as you upload this file, you will see an error as shown in Figure 1-4. The code *NoSuchWebsiteConfiguration* is a clear indicator that we have not enabled web site hosting.



404 Not Found

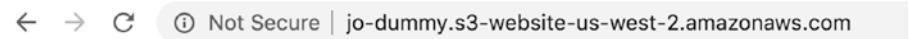
- Code: NoSuchWebsiteConfiguration
- Message: The specified bucket does not have a website configuration
- BucketName: jo-dummy
- RequestId: 716DA9971A5551D2
- HostId: nXEZVnKBos8pXNIQ635/Zni58xfs305K7FqdOtWbuZEOZ69CD/F9b3EzdOdItvPQs+ND+PsgeUY=

Figure 1-4. Error if Static Web Site Hosting Is Not Enabled

To allow hosting, switch to the **Properties** tab of the bucket, where you can find a tile titled Static Website Hosting. Click the tile and select the radio button “*Use this bucket to host a static website*” and type *index.html* in the field for index document and click Save. Hold on to your buckets; we are not done yet. If we reaccess the same URL, you will now see a **403 AccessDenied** error. We need to create a policy for the bucket to allow public get-object requests on this bucket. Every time a browser tries to fetch a file from this URL, the browser makes a get request for the object specified in the URL. To do that, go to *Permissions > Bucket Policy* and enter the following policy. Replace *jo-dummy* with your bucket name:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "PublicReadGetObject",  
            "Effect": "Allow",  
            "Principal": "*",  
            "Action": "s3:GetObject",  
            "Resource": "arn:aws:s3:::jo-dummy/*"  
        }  
    ]  
}
```

After what must have felt like a billion instructions, we are finally done, and now access your static web site URL to see the fruits of perseverance. Simple Storage Service says Hi as shown in Figure 1-5; give yourself an appreciative pat on your back.



Hello from Simple Storage Service

Figure 1-5. Successful Hosting of Static Web Site

Command Line Interface

I'm sure you have seen a flaw in the entire modus operandi. Any static web site beyond a hello world comprises more than just one file. There can be many HTML files, JavaScript files, style sheets, images, and many more. We don't want to upload each file separately using the web GUI that only allows uploading only one file at a time. Here comes the CLI for our rescue. We can upload files as a batch from our command line or sync an entire folder to a bucket with just one command.

For us to use CLI, we have to install it first. Installation instructions vary for each OS. For Windows, you can either download MSI installer or use Python pip. For Mac, Linux, and Unix, the following commands install the AWS CLI at `/usr/local/bin`. You need to have Python installed on your computer already:

```
$ curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"
$ unzip awscli-bundle.zip
$ sudo ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws
```

If you look closely at the first command, you can see that even the installer is being served from S3 directly. Once the installation is complete, add the CLI installation folder to your environment path variable to simplify command execution. You can find the detailed installation instructions and how to add it to the path variable for each OS at the following URL:

<https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html>

What we installed is AWS CLI which also includes S3 CLI, so we don't have to do this installation for each service. To make sure our installation is successful, run "**aws --version**" in the command line. You should see the version of aws-cli.

However, we have one more configuration to do. How can AWS recognize that the commands we run from our computer belong to the AWS account we created? It is time for us to wield our secret weapons, Access Key ID and Secret Key, we downloaded earlier when we created the admin user. To set up the default credentials, run the following command in the console:

\$ aws configure

This command asks you to give access key id, secret key, and region. Give the values corresponding to your IAM user admin. The default profile is used when we execute the CLI commands. If you want to use a different user like s3_publisher you created in the exercise earlier, run the following command and give the access key id and secret key of the user:

\$ aws configure --profile s3_publisher

By creating separate profiles, you can switch the CLI command's access levels without having to reconfigure every time we need to run a command as a user with a different set of permissions. With this, we successfully configured the command line interface. You can read about AWS CLI by executing command ***aws help***. All aws-cli commands take the following form:

```
$ aws [options] <command> <sub-command> [<sub-command> ...]  
[parameters]
```

The arguments mentioned in brackets are optional, while the ones in angular brackets are mandatory. Command parameter is to indicate which AWS service we want to interact with; or sometimes it can be a general command, for example, help. When we work with S3, the command takes the form of ***aws s3 <sub-command>***. The sub-commands vary depending on the command. For example, the cp sub-command intended to copy a file from the local file system to a bucket is applicable for S3, while it does not apply to the IAM service which doesn't have any file-based operations. To get subcommands applicable to each command, you can run command ***aws <command> help*** or even ***aws <command> <sub-command> help***.

Earlier, I promised that you could sync your entire folder with an S3 bucket. It can be done using the following command. It is easy enough to understand. The command is s3, and the sub-command is sync. The following two parameters --profile and s3_publisher tell aws-cli to use the credentials configured with profile name s3_publisher. The parameter --delete instructs the command to delete if a file in S3 doesn't exist in the local folder. All the files in a folder named dist are copied to an S3 bucket named jo-restaurant:

```
$ aws s3 sync --profile s3_publisher --delete ./dist s3://jo-  
restaurant
```

We can also perform operations like create-bucket, but as this is not a repetitive operation and has many input parameters, it is better if you create buckets in the AWS console.

DynamoDB

If we move away from the static file territory and want to deal with a bit more dynamic data, we can use the NoSQL database service DynamoDB. If you are not aware of NoSQL, the theory is elementary and overlaps with an RDBMS in some aspects of data storage. There are tables just like in relational databases. We insert records/items into the table; they are analogous to rows in a relational database. A row in a relational database consists of columns; however, they are called attributes or properties in DynamoDB. Unlike a relational database, we cannot have relationships between tables or foreign keys in DynamoDB. We can have triggers and indexes, but the approach is slightly different.

Each record in a DynamoDB table contains a partition/hash key and sort/range key and any number of attributes. There is no fixed schema except for the keys. Partition or hash key is analogous to a primary key in an RDBMS table. If we want to store multiple records with the same hash key, they must at least have different sort key values. From a relational database point of view, it is like having a primary key that consists of two columns. Apart from keys, there is no restriction on what attributes (or columns in relational database terminology) each record can contain.

We query the records using hash and sort keys for quick retrieval of records. If we want to find records using any other columns, it is a slow and costly operation. So there are certain cases where DynamoDB fits the bill and certain cases you are better off with a relational database. As long as you follow best practices of DynamoDB, you always get data in less than 5 milliseconds irrespective of how big the table is; the same cannot be said for an RDBMS. Another concept you need to learn about DynamoDB is its pricing. DynamoDB charges you for the provisioned throughput which

instructs DynamoDB to auto-scale so it can serve concurrent requests. If you are expecting 100 users today and 100,000 users tomorrow, all you need to do tomorrow is to increase provisioned capacity and reduce it once the high-volume traffic reduces. DynamoDB seamlessly scales up and down in a matter of minutes with the help of a few clicks. Try that on an RDBMS; you would appreciate DynamoDB despite its limitations.

Creating Tables and Insert Items

Open the DynamoDB service from the AWS Management Console. By now, you must be accustomed to the AWS organizational scheme. We can access each service through the console, so the next time, I need not repeat how to open an AWS service.

When you open DynamoDB for the first time, it presents you with a choice to create a table. Once you create a table, it will redirect you to the dashboard view. Click the Create Table button to open a new table creation wizard. Give the name as **orders** and partition key as **user_id** and sort key as **order_time**. Leave string as the data type for both fields. At the time of writing this book, DynamoDB does not have any data type for storing date or time. It is a common practice to store the timestamp in YYYY-mm-dd HH:mm:ss format. This way, you can filter out records of the same user_id by using the begins_with clause on order_time which is a sort key. One rule of thumb while querying for a specific record in a DynamoDB table is we should always pass full hash key and can apply conditions like *greater than, less than, and begins with, between* on a sort key.

You can leave the default values for table settings. If you want to drill down into the settings, deselect “Use default settings.” The UI presents you with an expanded table creation wizard wherein you can add secondary indexes. Secondary indexes are useful if you want to query using attributes other than hash key or sort key. The primary index, which is created by default, is a combination of hash key and sort key. The next setting is Read capacity units and Write capacity units. These dictate provisioned throughput and influence how many records we can read or write concurrently. The

higher the value, the higher the cost, and more simultaneous transactions are allowed. It all depends on what you need. The next setting, auto-scaling, is an automated way to increase table throughput to accommodate the higher load. Finally, Encryption related options are for keeping the data safe and secure.

Leave all these at default values and click Create. You will be redirected to tables view which shows that a table is being provisioned. In a minute or two, your table will be ready to be used. Navigate to the Items tab where you can see the items in the table. As you can guess, there will be none at the moment. You can click Create Item to insert one. It brings up a **Create Item wizard** which has two fields we have mentioned while creating the table. It should look as in Figure 1-6.

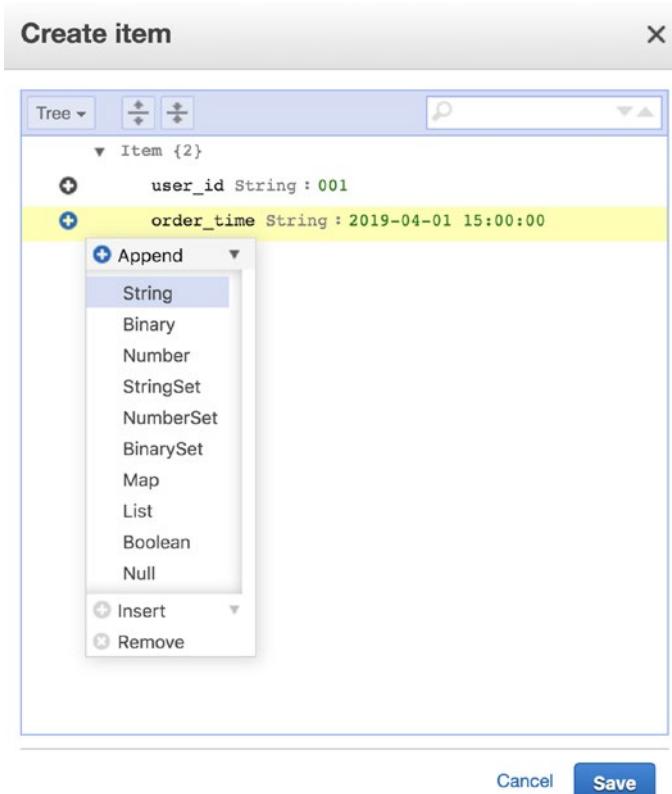


Figure 1-6. Create Item Wizard of a DynamoDB Table

Click the plus icon beside existing attributes to append or insert attributes. You can create an item with any number of attributes. As mentioned earlier, there is no schema you need to adhere to except for hash key and sort key. You cannot leave those values as blank. If you click without giving the values or if you create an item with the same hash and sort key as an existing record, the wizard complains. Give some values and click **Save**. You can find the record immediately in the Items tab.

Finding Data

If you want to query an item, you can do so from the Items tab. You may have to first switch the value in the dropdown that says **Scan** at the moment. There are two methods to get items from a DynamoDB table. The preferred method is to query if you already know the hash key of an item. If you don't, you can either scan or query in some other index's hash key. For this table, we haven't created any secondary index, so we can only scan or query. It should look as shown in Figure 1-7 if you choose Query.

The screenshot shows the AWS DynamoDB console interface for the 'orders' table. The 'Items' tab is selected. A search bar at the top contains the query: [Table] orders: user_id, order_time. Below it, the 'Query' dropdown is set to [Table] orders: user_id, order_time. The 'Partition key' section shows 'user_id' as a String type with a value of '001'. The 'Sort key' section shows 'order_time' as a String type with a placeholder 'Enter value'. Under 'Sort', the 'Ascending' radio button is selected. Under 'Attributes', the 'All' radio button is selected. At the bottom, a table displays one item: user_id 001, order_time 2019-04-01 15:00:00, and total_amount 14.55.

| | user_id | order_time | total_amount |
|-----|---------------------|------------|--------------|
| 001 | 2019-04-01 15:00:00 | 14.55 | |

Figure 1-7. In a DynamoDB Table

As you can see in the figure, you can provide two conditions for querying. Partition/hash key must be provided, and it is an equality condition. It should match the exact **user_id** you have used when creating the item. You can change the condition for sort key. If you don't provide any value for sort key, it returns all the records with the given hash key. You can also add a filter to further eliminate items from the query results.

Alternatively, if you switch to Scan operation, you can add any field in the filter condition. You may think that this is powerful than Query. Yes, it is if you don't know the hash key. However, at the same time, it is super slow. Because DynamoDB doesn't use any indexes for a scan operation, it has to go through every record and see if that record matches your filter and return it. One crucial point you must keep in mind is for every record it visits, it consumes read capacity, thereby reducing simultaneous scan operations that can be performed while the table is under operation.

You may wonder what happens if we consume all the throughput and one more request comes for finding an item. The new request gets throttled, and an error is returned. If you set up auto-scaling or on-demand provisioning, that may trigger increased capacity provision, or you will end up seeing errors in your application saying that request got throttled. Now, you see why scan operations must be used as a last resort and never on a live table that is serving a lot of user traffic. So find a way to perform a query all the time. Otherwise, go back to the design of your table and add a new secondary index or something.

Obtaining AWS SDK

When it comes to finding data, the most useful method is to find it programmatically unless you wish to hire a database assistant to query for records by your application every time. We can also do it using CLI as we used it with S3; but as I mentioned, the programmatic finding of data is more useful than CLI, so let us now see how to use AWS SDK to query for data.

However, when we speak about SDK, programming languages come into play. I cannot cover the nitty-gritty of each programming language SDK, so we have to pick a programming language in the book. However, we have provided multiple programming language coding examples in the source code that accompanies the book. In this instance, let us take NodeJS as an example of AWS SDK usage. However, any environment follows these steps at a high level:

1. Install the runtime environment (JDK and JRE/
NodeJS/Python/.Net/Go and many more)
2. Get the corresponding AWS SDK from the page -
<https://aws.amazon.com/tools/>.
3. Write and run the code to rule the world.

I do not want to go into the details of installing runtimes. However, you pick one of the platforms or programming languages. You should be able to follow along when you combine it with the downloaded source code if your runtime is not NodeJS. Installing NodeJS can be done by following instructions in the following URL:

<https://nodejs.org/en/download/>

Once you have NodeJS installed, you can install AWS SDK by running the npm install command as shown in the following. This will install the AWS SDK as a node package that can be used in your JavaScript programs. If you get any issues with installing it, you can refer to the documentation regarding your SDK in Amazon docs web site:

\$ npm install aws-sdk

If you are using Java runtime, you can get the dependency as a maven or gradle dependency, or you can download the jar file from the tools page of AWS given in the preceding text and add it as a dependency for your Java project. For Python, you can install **boto3** using pip. Other environments have similar mechanisms for getting the corresponding SDK.

Using SDK to Insert Data

Once the SDK is obtained, you can start incorporating it in your code. Irrespective of the programming language, a program using SDK consists of three parts as shown in Figure 1-8. Proving that you are the right party to run the code, create a client object of the AWS service in question and call the API methods to interact with it.

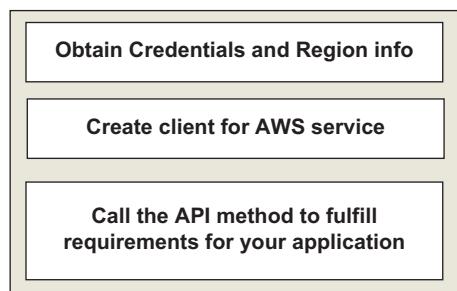


Figure 1-8. Typical Structure of a Program Using AWS SDK

First, you have to authenticate yourself using Access Key ID and Secret Key. Alternatively, we can use the Cognito service's identity pools. As I previously mentioned, sharing access key id and secret key is not a safe way to interact, especially if you are hard-coding them in JavaScript that gets executed in the browser. Any technically savvy user can get hold of those credentials and can use them in his/her programs to get access to your cloud resources. For your understanding, I will show both ways to access credentials; however, be mindful of your usage.

To use Cognito, open the Cognito service from the AWS Management Console and click the button that says **Create new identity pool**. Give a name for the identity pool and select the **Enable access to unauthenticated identities** checkbox for keeping things simple for this exercise. The steps are depicted in Figure 1-9.

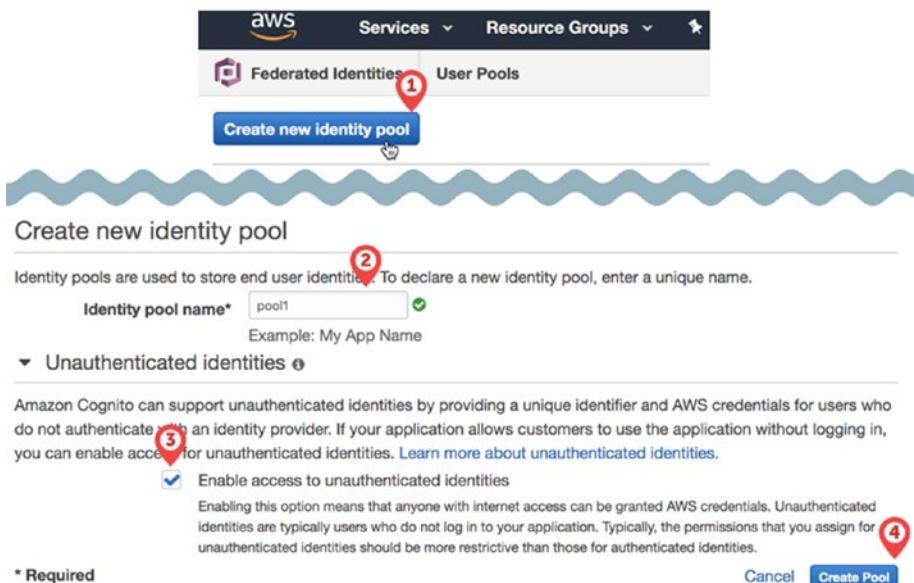


Figure 1-9. Steps to Create an Identity Pool

This creates an identity pool and creates two IAM roles – one for authenticated users and another for unauthenticated users. By selecting this, we want to avoid login operation to be able to use SDK. However, we are using unauthenticated identity, to keep the complexity levels low for now. When you are developing real applications, make sure unauthenticated user has the least permissions given through IAM.

The unauthenticated role can be used to give many restrictive permissions like read-only access to DynamoDB or a specific S3 bucket. Authenticated users can be allowed to gain access to this role by a login

operation, which is out of the scope of the book but is simple enough if you follow the instructions in the subsequent pages after creating an identity pool. It can be achieved through a few lines of JavaScript. If you want to allow usage of unauthenticated role in a JavaScript, the following code snippet achieves that. The IdentityPoolId should be replaced with the ID of the pool you just created:

```
// Initialize the Amazon Cognito credentials provider
AWS.config.region = 'us-west-2'; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
    IdentityPoolId: 'us-west-2:fa17c899-4319-4048-a8ff-
        fa2a7fd73868',
});
```

As soon as you create an identity pool, you will be given the option to copy the sample code. The options include Android, iOS, .Net, and a few other platforms that can use these Cognito credentials. The preceding snippet is for an id I just created, which I will have already deleted by the time you read this book, so the code snippet does not work for you unless you change the region and id as per your pool. If you want to hard-code the credentials instead of using a Cognito pool, the following snippet shows how to do it:

```
AWS.config.credentials = {
    accessKeyId : "AKIAJHM6N3MGEYXBKEHA",
    secretAccessKey : "YR8BP2z5PpX/
        Ferg+T0q1sT8JfZP8H6eCdm1HhRN"
};
```

Even the region information is better kept out of source code because you may want to run the code in machines closer to the client using it. Hence, you will gain better performance by servicing it through some form of configuration files that are specific to each region's hosts. If you are hell-bent on keeping your application less secure, you can hard-code

the accessKeyId and secretAccessKey in your source code as shown in the preceding text. Remember that this is as good as leaving Pandora's box wide open.



Caution Avoid hard-coding Access Key ID and Secret Key in server-side programs. Obtain them through environment variables or such. For client-side JavaScript, use Cognito.

Once authentication and region configurations are completed, you can use that configuration to create the necessary client. Each AWS service has its own client. Create an object of the client, and you will be able to make calls to that service by invoking functions on that object. These two steps do not change irrespective of how we have set credentials. Creation of a DynamoDB client is shown in Listings 1-1 and 1-2.

Listing 1-1. NodeJS way of creating a DynamoDB client

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({apiVersion: '2012-08-10'});
```

Listing 1-2. Creating DynamoDB client in an HTML file

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.410.0.min.js">
</script>
<script type="text/javascript">
    // Create DynamoDB service object
    var ddb = new AWS.DynamoDB({apiVersion: '2012-08-10'});
</script>
```

Both mechanisms create DynamoDB objects by using AWS namespace. Mechanisms of how they are injected may vary, but once we have the AWS object available, the code is pretty much the same for setting credentials and performing operations on DynamoDB tables and data. Let us first see how to perform insertion into an existing table.

Here is the snippet to insert data in DynamoDB table named **orders** that has user_id and timestamp as attributes as well as keys. We can add any number of attributes as long as we provide values for keys. I do want to reiterate the fact that DynamoDB doesn't insist on any schema. However, you have to pass the data type along with attribute value. That is what you see as **S:** and **N:** in front of the values. It means a string and number, respectively. Listing 1-3 demonstrates how to do just that.

Listing 1-3. Inserting an item into a DynamoDB table

```
var params = {
    TableName: 'orders',
    Item: {
        'user_id' : {S: '001'},
        'order_time' : {S: '2019-04-02 14:00:00'},
        'total_amount' : {N: '10.5'}
    }
};
// Call DynamoDB to add the item to the table
ddb.putItem(params, function(err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success");
    }
});
```

The second half of the previous snippet shows how to insert the item into the table. We have to call the putItem method on the client object we created earlier. Listing 1-4, as shown in the following, is the complete code of the example. You can save this as any file with extension .js and can run it with the node command. Alternatively, you can include this code in an HTML file script tag to see the same result when you open it through a browser.

Listing 1-4. chapter-1/demo-01/insert-record-dynamodb.js

```
var AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
    IdentityPoolId: 'us-west-2:fa17c899-4319-4048-a8ff-
        fa2a7fd73868',
});

var ddb = new AWS.DynamoDB({apiVersion: '2012-08-10'});

var params = {
    TableName: 'orders',
    Item: {
        'user_id' : {S: '003'},
        'order_time' : {S: '2019-04-02 14:00:00'},
        'total_amount' : {N: '10.45'}
    }
};

ddb.putItem(params, function(err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success");
    }
});
```

In the sources provided with the book, you should be able to find both examples by going to the chapter-1 folder. Along with JavaScript, you can find Java and Python examples as well. For the sake of providing the context, let me take Java and Python sources, as shown in Listings 1-5 and 1-6, and explain them as well. You have to add AWS SDK dependencies for these code snippets to compile. I have provided maven pom.xml so that you can get the dependencies by running the **mvn install** command.

Listing 1-5. chapter-1/demo-01/InsertRecordDynamoDB.java

```
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.*;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import java.util.HashMap;

public class InsertRecordDynamoDB {

    public static final String TABLE_NAME = "orders";

    public static void main(String... args) {

        ProfileCredentialsProvider provider = new Profile
        CredentialsProvider("admin");

        final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder
            .standard()
            .withCredentials(provider)
            .build();

        HashMap<String, AttributeValue> item_values = new
        HashMap<>();
        item_values.put("user_id", new AttributeValue("789"));
        item_values.put("order_time", new
        AttributeValue("2019-02-29 00:00:00"));
    }
}
```

```
    item_values.put("total_amount", new
AttributeValue("11.95"));

ddb.putItem(TABLE_NAME, item_values);
}
}
```

Listing 1-6. chapter-1/demo-01/insert_record_dynamodb.py

```
import decimal
import boto3

session = boto3.Session(profile_name='admin')

dynamodb = session.resource("dynamodb", region_name="us-west-2")
table = dynamodb.Table('orders')

item={
    'user_id': "001",
    'order_time': common.get_time(),
    'total_amount': decimal.Decimal('7.89')
}

table.put_item(Item=item)
```

In this Java snippet, we are pretty much following the same format. We are getting the credentials from the profile we configured using the **aws configure** command earlier. We are then creating a DynamoDB client using the builder provided by AWS SDK for Java. We then construct a map object with column names for key values. Once the object is constructed, we can call the putItem method of the client we have constructed.

The Python snippet also is pretty similar. We use the boto3 library and create a session object using the same admin profile we configured. Then we create a DynamoDB resource and eventually construct a table object from it. The table object has a put_item method that can be used to insert

records. Feel free to change values and run the code several times. If you make any mistakes like giving a wrong profile name, the programs throw exceptions that are easy enough to understand.

Now we got the similarities across programming languages out of the way; we will be showcasing only one programming language in the book. For the remaining programming languages, you can refer to the source code. The source code is organized chapter-wise and demos-wise. The preceding snippets to insert data in a DynamoDB table can be found in the chapter-1/demo-00 folder for each programming language.

Querying Data with SDK

We have seen how to insert data in a DynamoDB table in the previous section. You should have realized that it is so easy to set up and work with databases. It barely takes 10 minutes from opening AWS console to creating a table and to insert records programmatically. However, inserting records is just half the battle. We need to find the data we want. To tell a user of our application that his order has been prepared and is on the way, we need to fetch his order information from the database. Let us see how to do just that.

In the world of DynamoDB, there are two ways we can get data: query and scan. The query is when we are fetching a record when we know its partition key at the least. If we don't know its partition key/hash key, the only way for us to find any record is to perform a scan operation. The scan operation involves reading every record in the table and seeing if it matches a condition you have provided. Even in a relational database, performing a WHERE clause on a non-indexed column is slow. The same applies for a scan operation in a NoSQL database. The hash key is an indexed column optimized for finding exact matches.

Querying through SDK also follows the same pattern as inserting. Get credentials using either Cognito or a preconfigured profile or hard-code the access key id and secret key. Using the credentials, create the client of

DynamoDB and perform query operation as shown in Listing 1-7. We can also create a table object to execute queries on it. The gist is that there are numerous ways to perform queries and the library keeps getting better.

Listing 1-7. chapter-1/demo-01/query-dynamodb.js

```
let AWS = require('aws-sdk');
let com = require('./common');

com.configure_credentials(AWS);

let docClient = new AWS.DynamoDB.DocumentClient();

let query_input = {
    TableName : "orders",
    KeyConditionExpression: "user_id = :id and begins_"
    with(order_time, :ts)",
    ExpressionAttributeValues: {
        ":id":"001",
        ":ts":"2019-04-01 1"           // Change this timestamp
                                         prefix so suit the time
                                         you insert
    }
};

// Call DynamoDB to query items from the table
docClient.query(query_input, function(err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log(data.Items);
    }
});
```

In the preceding code snippet, we moved setting credentials to a different file named common.js and called the method to keep our code snippets clean. Another change is that we have not created an object of the DynamoDB service; instead, we created an object of a DocumentClient. It takes a simplified approach to query the data. To query, we have to construct a **QueryInput** object that encapsulates table name, key conditions, any filter conditions we may want to pass, and so on. Keep a note of the begins_with clause used on range key. If we don't provide any range/sort key condition, query operation will return all the items having the same hash key. Along with that, the query input object allows us to control which parameters must be returned as well.

Another aspect you can make a note of is the parameterization of values. **:id** and **:ts** for attribute values are parameters whose values are provided in the **ExpressionAttributeValues** object. We can also parameterize column or attribute names, but it would be only necessary if we have a hyphen (-) in the name of the column or any other special names like keywords. The query input object will look like the following in that case. Observe that the attribute names are parameterized using a # and the names are provided in the **ExpressionAttributeNames** object. By using this, you can use column names with a hyphen or a keyword:

```
let query_input = {  
    TableName : "orders",  
    KeyConditionExpression: "#id = :id and begins_with(#ts, :ts)",  
    ExpressionAttributeNames: {  
        "#id": "user_id",  
        "#ts": "order_time"  
    },  
    ExpressionAttributeValues: {  
        ":id": "001",  
        ":ts": "2019-04-01 1"  
    }  
};
```

Lambda

We have dipped our toes into the cloud. It's time for us to get deeper by taking a cloud-based approach to executing our programs. AWS Lambda is a serverless cloud computing platform. It is event driven which means we can use the service to run code when preconfigured events happen. Best of all, we do not have to have any servers; that is what serverless means.

The term serverless is an obvious misnomer because there are servers running somewhere in AWS data centers, but they are cleverly time shared to run millions of lambda functions from different users without stepping into each other's territories. The reason it is called serverless is that you don't have to configure where it runs and you are not billed when your code is not running. The pricing model of Lambda is based on how much RAM you use each time you execute a lambda function and how many milliseconds you run the program. So, if you build a lambda function to save feedback from a static web site, you are only charged when someone posts feedback from that web site.

There is a generous amount of free tier for a lambda function. First, 1 million execution units are free. It amounts to 400,000 GB-Seconds. So, if you choose 1 GB RAM for each invocation, you can invoke your lambda function for 400,000 times each month for free. That is a lot of compute. However, keep an eye on other costs as well. Like data transfer costs and CloudWatch logs, if you wrap your lambda in an API gateway, those calls also include some amount of cost. In any case, to work on the coding examples of the book, the free tier should be sufficient.

Creating a Lambda Function

Let us open the Lambda service from the AWS console. Make sure that you are in the correct region. This information is displayed in the top-right corner of every page in AWS. We shall now create a Lambda function that interacts with the **orders** table we created in the previous section. So both

the Lambda function and DynamoDB table must be in the same region for smooth interaction. Once you are in the Lambda service, click the **Create a function** button to open a **Create Function** page.

You can create a Lambda function from scratch; it is the default option. If you want to see some examples or quick start from a different blueprint, you can choose one of the other two options instead of **Author from scratch**. They do simplify some access related to configurations, but I want you to see what goes under the hood of these blueprints.

The basic information section of the page is for you to give a name to the lambda function and choose a runtime. Give **orders_lambda** as the name of the function. You can choose any programming language listed in the runtime dropdown or go for a **custom runtime** which allows you to make use of programming languages other than the ones listed. Let us not do anything fancy for our first foray into Lambdas. Stick with NodeJS or Java or Python for which you have written DynamoDB-related code following the previous section's instructions. Finally, you have to give permissions for your Lambda function to be able to interact with DynamoDB or S3 or whatever service you want to invoke through our Lambda. Choose **Create a new role** in the permissions section of the page and click Create Function.

You may be overwhelmed by the number of configuration options displayed on the screen as shown in Figure 1-10. Remember that the screenshot is not showing all the configuration options. The screen is divided into many sections. Let us only focus on the key sections we need to use. The first portion is the top sticky header. It contains options related to versioning the code of lambda function, testing the code, and throttling configuration. Right below the sticky section, you can find **Designer** in the Configuration tab. The designer is for us to configure triggers, layers, and the resources this function has access to.

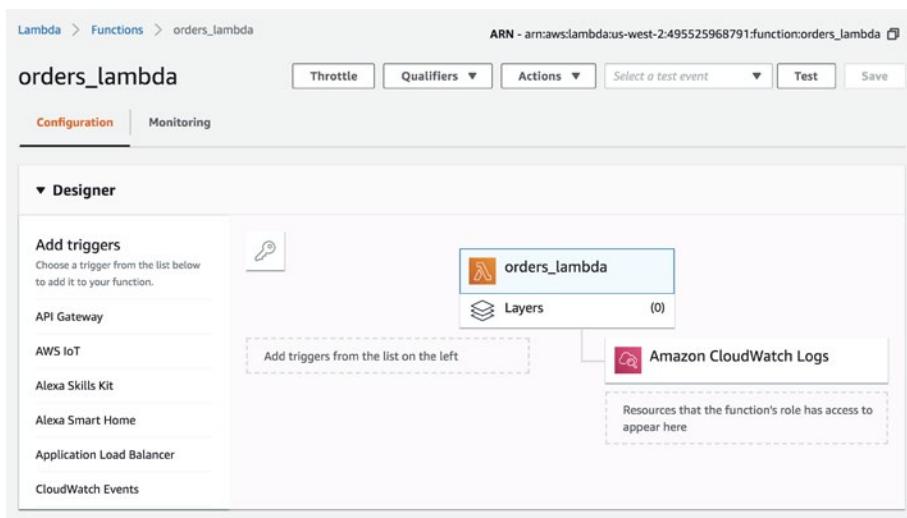


Figure 1-10. Lambda Configuration Portion

Triggers are for configuring when this lambda function is invoked. If you want to trigger this function through a Rest API or call from an IoT device or when a row gets updated in DynamoDB, you can mention that here. For our use case, we don't need it. Layers are for managing common code shared across multiple lambda functions. Right now, our lambda function can only publish logs to the CloudWatch service, and it has no other access. We need to give DynamoDB access to our lambda function. I will show how we can do that in a moment. If you scroll a little further down, we will find an IDE to write code in. However, this does not apply to all the runtimes. Only scripting languages like JavaScript and Python support cloud IDE.

Here, we need to code for what happens when the Lambda function is triggered. Move to the code section. We should have a file named **index.js** which is a starting point. We need to provide an implementation for **exports.handler** because the handler field just above the IDE mentions **index.handler**. If you create a new JavaScript file, you should provide the file name instead of the index and the method you have exported in that file.

Time for us to understand the code. The default code is a HelloWorld template. It just constructs a response JSON and returns it. You can write your own implementation, but you should eventually return a response object.

Let us see how to run the function. Notice the sticky section that has a button **Test**. You have to select a test event. It is a way for us to mock the trigger event and pass the input to the lambda function. By default, the test event is empty except for an option to configure test events. If no events are configured so far, a new “**Configure test events**” wizard will pop up. You can select an event template if you are testing for a specific event trigger. For example, you can select Amazon SQS to see a template JSON how input is passed to the lambda function. Go back to HelloWorld template; it just contains some key and value pairs. You can leave them like they are and save them with any name. The following snippet shows how to access data passed in as input:

```
exports.handler = async (event) => {
    console.log("Value for key1: " + event.key1)
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda1!'),
    };
    return response;
}
```

Now, you can click the **Test** button again to run the lambda function. The execution result is displayed in the IDE or below the Test button. It depends on where you are in the page right now. If you have scrolled down to IDE and run the test, it shows the result in the IDE, or if the IDE is not visible, the result will be at the top. Observe that the response also contains information about how many milliseconds it ran for and what is the billed duration. The minimum amount of billed duration is 100 ms. At 128 MB memory, it is 0.0125 GB-Seconds. At this rate, you can make 31 requests per second for the whole month, and you will still be inside free tier limits.

Accessing DynamoDB from Lambda

The previous section showed how easy it is to run code using Lambda without even having any server. Let us take it forward and see how to access DynamoDB using this lambda function. In our example, we will be passing input as user_id and total amount and store them in the database. To do that, we first have to give our Lambda function permission to access DynamoDB. Scroll further down to reveal other tiles related to configuration. What we are looking for is a tile that is titled **Execution role**. Here it shows a role created for our lambda function as shown in Figure 1-11.

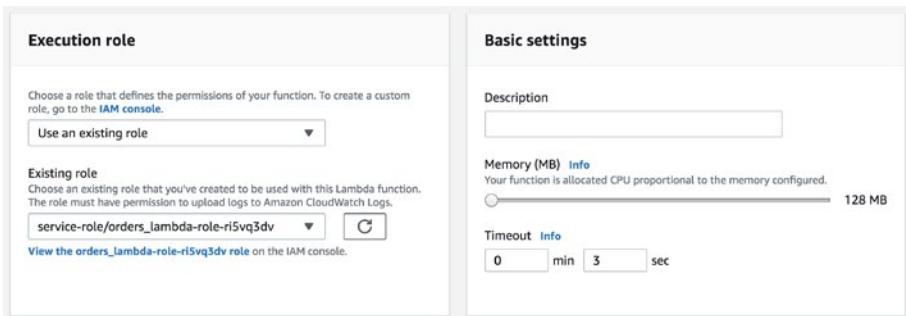


Figure 1-11. Role, Memory, and Timeout

Similar to an IAM user, an IAM role is used to permit specific operations in AWS, but roles don't have any credentials associated with them, and any user can assume a role, thereby gaining access to the necessary AWS service. Click the link just below the service role dropdown to go to the IAM service and give necessary permissions to the IAM role assigned to this lambda function. As I mentioned, we would want to give access to DynamoDB similar to how we have given access to a user we created in the IAM section of this chapter. If you want a quick refresher, click the Attach Policies button and search for '**dynamo**' and select the policy named **AmazonDynamoDBFullAccess** and save. It gives the permission required for your Lambda to access database tables we created

earlier. If you scroll back to the Designer tile of your lambda function, you should see Amazon DynamoDB listed below Amazon CloudWatch in services that the role of this lambda function has access to.

The other tile in Figure 1-10, Basic settings, is to control how much memory a Lambda function has and how long should this lambda function run. By increasing the amount of memory, you would also be increasing CPU capacity allocated to each lambda invocation. The amount of memory you would need depends on what your code is doing. Lambdas are supposed to be doing a small amount of work and terminate once finishing their jobs. Don't expect them to run for hours. Because lambdas are charged based on how long they execute, this timeout controls the functions that went into an infinite loop emptying your bank balance.

The code for inserting the data in the orders table is almost the same as our local NodeJS version, except for the fact that we read the input from the event input parameter instead of hardcoding it. Replace the code of your lambda function with the code in Listing 1-8.

Listing 1-8. Inserting items in a DynamoDB table

```
const AWS = require('aws-sdk');

exports.handler = (event) => {
    let ddb = new AWS.DynamoDB({apiVersion: '2012-08-10'});

    // Get timestamp in YYYY-MM-DD HH:mm:SS.sss format
    let ts = new Date().toISOString().replace('T', ' ')
        .replace('Z', '');

    let params = {
        TableName: 'orders',
        Item: {
            'user_id' : {S: event.user_id},
            'order_time' : {S: ts},
            'total_amount' : {N: event.total_amount}
        }
    }
}
```

```
        }
    };
    console.log("inserting data");

   ddb.putItem(params, function(err, data) {
        if (err) {
            throw err;
        }
    });

    return {
        statusCode: 200,
        body: "Successfully Inserted Record",
    };
}
```

This code reads user_id and total_amount from the input and inserts the data into DynamoDB using the same putItem method we used for native NodeJS code. For this code to work, you have to click Configure test events and change the input data when you test. You can use the following input with the same key names as in code:

```
{
    "user_id": "101", "total_amount": "10.34"
}
```

However, I want to bring a few other items to your notice. We don't need to fetch and set AWS credentials as we have already given the permissions for this lambda function's role through IAM. We are also generating a new timestamp every time we run this code, so if you run this code for five times, you should see five records in the database. Each of them should have the same user_id but have different timestamps.

You can try to change the data type of the timestamp parameter from string to number by changing that line to **'order_time': {N: ts}** and try to run the same lambda function. This invocation throws an error saying that **"The parameter cannot be converted to a numeric value."** You should see that in the log.

You should be able to change the code to read the data based on your previous code snippets from the DynamoDB section. Just ignore the part where you have to read the credentials using Cognito or a preconfigured profile.

Let us now see how the process differs if we have to develop a lambda function in Java. Python code can also be edited in the cloud IDE of Lambda, but Java code can't be edited like that because it involves compiling the source before it can run. I have included a sample code, as shown in Listing 1-9, for writing a Lambda function using Java. You can build a jar of your project including all the dependencies like **aws-sdk** using either maven shade plugin or your IDE capabilities to generate a JAR artifact and upload it. If you don't include any dependency, you will get *NoClassDefFoundError* when executing the Lambda function. In the sources provided to you, I have also included a pom.xml in the demo-02 folder of chapter-1 inside java-sources. It helps you to build a jar file that can be uploaded to the Lambda function directly. You can run the **mvn package** command to generate the jar.

Listing 1-9. chapter-01/demo-02/SaveOrderLambdaHandler.java

```
import com.amazonaws.services.dynamodbv2.*;  
import com.amazonaws.services.dynamodbv2.model.AttributeValue;  
import com.amazonaws.services.lambda.runtime.Context;  
  
import java.time.LocalDateTime;  
import java.time.format.DateTimeFormatter;  
import java.util.HashMap;
```

```
public class SaveOrderLambdaHandler {  
    public String handleRequest(Order input, Context context) {  
  
        final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.  
        defaultClient();  
        String ts = new AttributeValue((LocalDateTime.now()  
            .format(DateTimeFormatter  
            .ofPattern("YYYY-MM-dd  
HH:mm:ss"))));  
  
        HashMap<String,AttributeValue> item_values = new  
        HashMap<>();  
        item_values.put("user_id", new AttributeValue( input.  
        getUser_id()));  
        item_values.put("order_time", ts);  
        item_values.put("total_amount", new  
        AttributeValue(input.getTotal_amount().toString()));  
  
        ddb.putItem("orders", item_values);  
        System.out.println("Item inserted successfully");  
        return "Item inserted successfully";  
    }  
}
```

Observe that I'm passing input as an order object. It is a POJO class I created for representing columns user_id, timestamp, and total_amount in a clear manner. The input is automatically parsed to the order object as long as keys of input are matching with the field name. Refer to the source code for Order class. Make sure that Handler field in Function code part of Lambda UI contains package, class, and method implementing the logic. If your code is in a method named **handleRequest** in a class named **SaveOrderLambdaHandler**, your handler field should be **SaveOrderLambdaHandler.handleRequest**. Once you fill in those fields, it should look like in Figure 1-12, and it follows a similar approach to NodeJS.

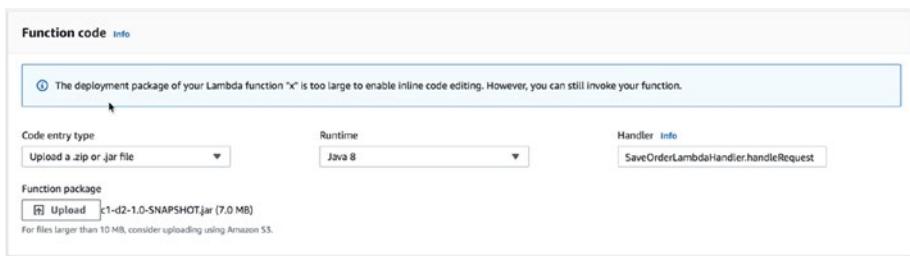


Figure 1-12. While Uploading the Jar, Ensure That the Handler Is Correct

The whole process of developing, creating JAR, and deploying to a Lambda function may appear cumbersome and not manageable on a large scale. To address this issue, AWS has released numerous toolkits for IntelliJ and Eclipse that simplify this whole process of developing and deploying lambda functions. AWS Toolkits for Eclipse and IntelliJ are provided as plugins of the IDEs. AWS SAM is a framework that also features many utilities for local testing and deployment of Lambda functions. However, these are out of scope of this book.

AWS Toolkit plugins for Eclipse and IntelliJ simplify Lambda development along with providing many other features for interacting with AWS right from IDE. SAM CLI is another framework that provides a lambda-like execution environment on your machine.

Invoking Lambda Function from an HTML Page

So far, we have been invoking the lambda function by clicking the Test button and configuring test events. It is useful enough for testing the lambda, but this is barely an option for a real-world application. There are various mechanisms we can use to invoke a lambda function:

1. Through a trigger such as SQS message, DynamoDB Stream, or CloudWatch event
2. Exposing the Lambda function as a Rest endpoint through API gateway
3. Invoking the Lambda function by using Lambda API from a client

Options 1 and 2 can be configured through the triggers portion of Lambda configuration. We will see how to invoke using the third mechanism. Any application can be a client application – a web page, a desktop application, or even a server running our application. For this approach, we have to use Lambda SDK. We have already made use of SDK for inserting data in DynamoDB. Invoking a lambda function is no different.

We get credentials of a user with permission to invoke a lambda function using either Cognito or a preconfigured profile; we use the credentials to create a Lambda client and call the invoke method. Save the following code in an HTML file and open it in a browser. Make sure you change the ID to pass your Cognito pool id. You can skip the rest of the HTML as shown in Listing 1-10.

Listing 1-10. Writing the code directly in HTML

```
<!DOCTYPE html>
<input id="customer_id" type="text" placeholder="Customer ID">
<input id="total_amount" type="text" placeholder="Total Amount">
<button class="btn default" onClick="insertOrder()">Insert
</button>
<p id="result"></p>
```

CHAPTER 1 GETTING STARTED

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.410.0.min.js">
</script>
<script type="text/javascript">

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.CognitoIdentityCredentials
({IdentityPoolId: 'ID'});

let lambda = new AWS.Lambda({region: 'us-west-2',
apiVersion: '2015-03-31'});

function setResult(text) {
    document.getElementById('result').innerText = text;
}

function insertOrder() {
    let id = document.getElementById('customer_id').value;
    let amount = document.getElementById('total_amount').
        value;
    let order = {
        "user_id": id,
        "total_amount":amount
    };

    let invokeParams = {
        FunctionName : 'orders_lambda',
        InvocationType : 'RequestResponse',
        Payload: JSON.stringify(order)
    };

    setResult("Invoking Lambda Function");
    lambda.invoke(invokeParams, function(error, data) {
        if (error) {
            setResult(error);
        }
    });
}
```

```
    } else {
        setResult("Completed" + (data.Payload ===
        "null" ? "" : ":" + data.Payload));
    }
});
</script>
```

You can give some values in the text boxes and click **Insert**; you may see a prompt complaining about not having access to invoke lambda function. I do want to remind you that the unauthenticated role we created earlier in the Cognito service only has access to DynamoDB. If we try to invoke a lambda function using the same identity, we are bound to see an error saying that the user doesn't have access to invoke a lambda function. So go to the unauthenticated Cognito role and give that user access to invoke Lambda function by attaching inline policy **AWSLambdaInvocation-DynamoDB**. It gives necessary authorization for the identity to invoke a lambda function.

Once the necessary access has been granted, give values in customer id and total amount text boxes and click Insert. If you give valid values, you should see a completion message; otherwise, you see some generic error message saying the process exited before completing the request. It is because we haven't done appropriate error handling in Lambda function.

Observing Logs in CloudWatch

We can always improve the lambda function to throw back meaningful error messages in case of errors while running the code. However, how do we read the full logs that were so helpful when we were testing the lambda function? How can we find logs of all our lambda function invocations? It is important for us to be able to debug production issues. Fear not. Logs are there; we have to open a service called CloudWatch to read the logs. The CloudWatch service is for collecting the logs and monitoring the health of our application.

To access logs generated by any lambda function, go to the Monitoring tab of the Lambda function as in Figure 1-13. This tab shows numerous graphs. You can see what is the average duration, how many times it was invoked, and what is the success rate. There are many more graphs if you scroll down in the Monitoring tab. To see the logs, click the View logs in CloudWatch button above the graphs.

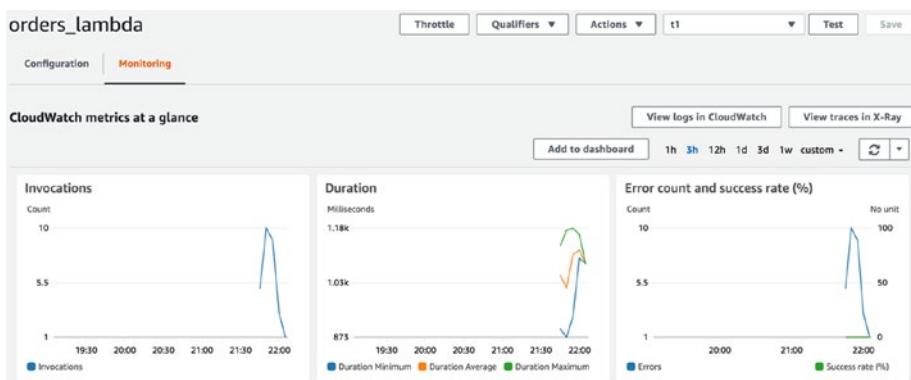


Figure 1-13. Monitoring the Lambda Function

It opens the orders_lambda log group. In the AWS world, we organize logs in log groups. Each log group can store logs for one or more applications based on how you organize. In our log group, logs are split into log streams. They are just time-sliced portions of log. Open the latest log stream to find the recent logs from the lambda function. You should also see the error you may have generated by giving invalid value or any log message we may have printed to the console in our lambda function.

There are a few other services that we can use to run code on the AWS cloud. Elastic Cloud Compute or **EC2** is a service that allows you to rent virtual computers and run whatever you wish. So, if you want to run an application that is not suitable to run serverless, you can create as many virtual computers as you want and deploy your application. For example, if you have developed a J2EE application that has to be up and running all

the time, you can create an EC2 instance and deploy your application in that once you have installed any web server of your choice like Tomcat or such. You can also use other services such as **Elastic Beanstalk** to simplify some of the tasks with launching web applications.

If you want to run applications that are containerized, you can use Elastic Container Service or **ECS** for that purpose. The containers run on one or more EC2 machines, but orchestration or management of the containers is taken care of by ECS. We will cover these services on the need basis. For us to start learning about Application Integration, we have enough basics covered. Let us understand the sample application we are going to be using to build features on using services in the Application Integration portion of AWS.

Sample Application

Throughout this book, we build an application for a restaurant. Customers can open the web site and place orders for the items they want. It is a simple application with minimal features required for us to focus on learning Application Integration. We shall develop the application from a basic static web site and add necessary features.

The starting point is no whiteboard either. You should be able to get the starting point HTML, JS, and CSS files from the folder *JavaScript-sources/awsome-restaurant-start*. It is a VueJS application on the NodeJS platform, so there is a slight learning curve if you have never tried VueJS. VueJS is like AngularJS or React but with a very small footprint and short learning curve. I shall explain the necessary concepts if you have never worked with it. Let us first run the following commands inside *awsome-restaurant-start* to start the application:

\$ npm install ← Only for the first time to get all NodeJS dependencies

\$ npm run serve ← To run application and enable hot reload

Once you run the second command, the application will start, and you will be shown URL to access it. The address will most likely be **http://localhost:8080**. Hot reload enables you to make the changes in source code, and the changes will reflect in the UI automatically. The application will look like in Figure 1-14.

In the home view of the application, you are presented with a few food items that you can order. When you open the application for the first time, the new user id is generated and saved in the local storage of the browser. The next time and onward, the same id is used. When you click Order on any food item, you order that item. Don't hold your breath for the food to be delivered; all it does is to trigger a Lambda function to save the order in the database. The order is also saved in local storage.

The other two views are place holders for future functionality we will try to bring while learning about how to use SQS, SNS, SWF, and Step Functions. We will not try to implement features like authentication or making the application look beautiful. We shall take shortcuts to save time for learning the core topics of the book.

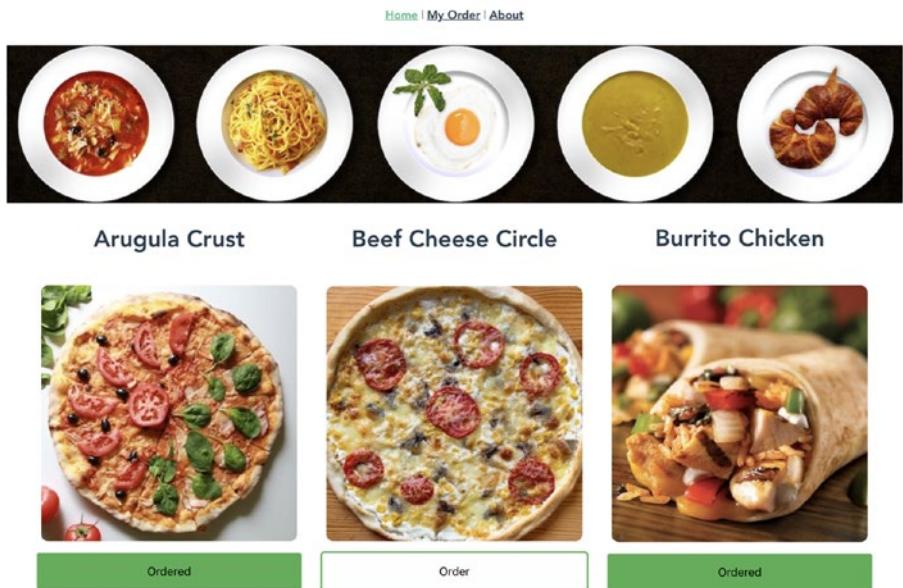


Figure 1-14. Sample Application Home View

Now you should understand why I have asked you to create the orders table and the lambda function to save items in it and so on. As I have already shown you how to invoke lambda, you should understand how the code in the sample application is doing that. We will deploy this application in S3 so that we can launch our first running application in the cloud. Create a new bucket in your AWS account with any name and enable static web site hosting, or you can use the bucket you have created earlier. Open the package.json file in the application and modify s3://awsome-restaurant to your bucket name in line 9. Run the following command to deploy:

```
$ npm run deploy
```

Provided that you have given all the necessary permissions for s3_publisher in the previous examples, all the files get pushed to S3. You can access it using the same S3 URL you have used to access a static web site we launched in the S3 section of this chapter.

Understanding the Project

When you have not run the application yet, it contains public and src folders and some configuration files to begin with. It is illustrated in Figure 1-15. The files in the public folder are static assets that don't change, like images, icons, and *index.html*. We can inject components and views into this HTML file using VueJS.

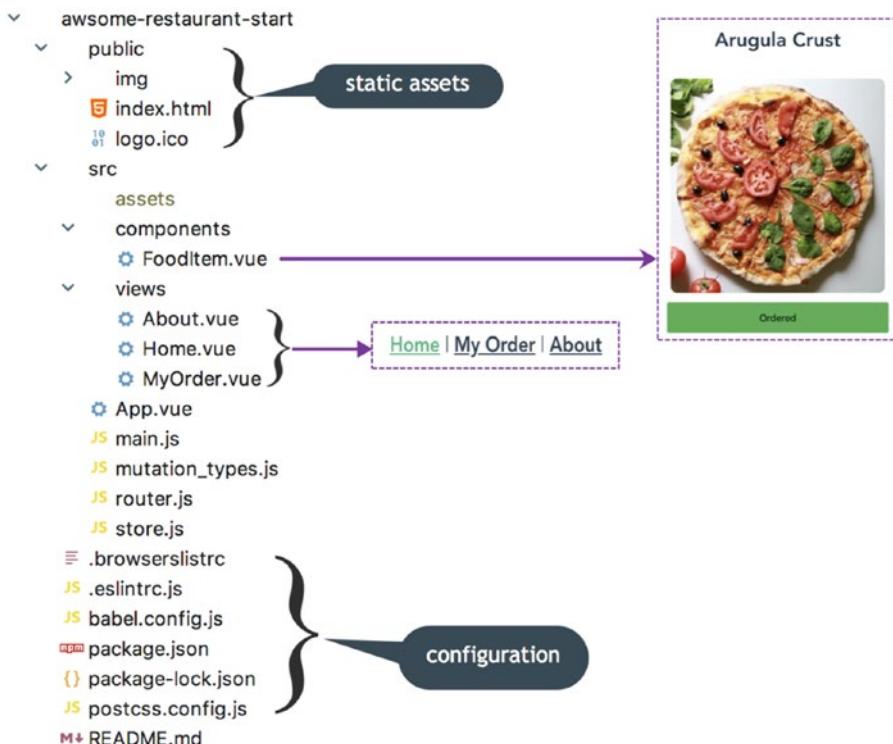


Figure 1-15. Project Structure and Organization

The folder **src** is for dynamic assets like JavaScript, components, and views. The files with extension '**.vue**' are Vue components. Each Vue component consists of an HTML template, styles, and script associated with it. The FoodItem.vue will render a food item based on the data provided to it. The code that should be triggered when we click the Order button resides in FoodItem.vue. This component is injected in Home.vue.

All the components in the views folder are for segregating different views of our application. We can switch views by clicking the links at the top of the application. App.vue is the main container that holds these views.

The JavaScript files contain central logic that doesn't belong to any component. For example, *store.js* contains logic to maintain common data such as user id and ordered items and also handles saving them to local storage whenever there is change. Other components use the store to access or modify data. The file *mutation_types.js* is just a constants file to represent different changes that can happen on the data stored in *store.js*.

The file *router.js* is for tying each view with a URL and components in the views folder. The file *main.js* brings all of these elements together. It combines router and stores and creates a Vue object and injects into a DOM element with id **app**. If you open *index.html* in the public folder, you will just see one div with id **app**.

The remaining files in the project folder are for configurations such as linting; js compile using babel. The file *package.json* lists all the node package dependencies for this project. When you execute the npm install command, this file is used to determine what is needed to be installed for your project to run. This file also has the deploy command you have modified earlier.

Vue Components

If you are already familiar with VueJS, you can skip this section. I shall give a crash course on what is inside a Vue component. Each component takes up the following format. It can be divided into template, script, and style:

```
<template>
    // Markup
</template>
<script>
    // Javascript code
</script>
<style>
    // Styles applicable for this component.
</style>
```

Vue.js requires an HTML-based template, and it allows you to declaratively bind the rendered DOM to the underlying Vue instance's data. If that sounds like a lot of jargon, let me explain. The Vue instance is an object we create inside the script tag of a Vue component. We can create some properties in here and use those properties in HTML tags inside the template. It effectively binds them together. Whenever the variable value changes, it updates the UI and vice versa. Take the following Vue component in Listing 1-11 for example.

Listing 1-11. javascript-sources/chapter-1/demo-04/awosome-restaurant/src/views/MyOrder.vue

```
<template>
<div class="my-order">
    <h1>{{ message }}</h1>
    <p>Click button to increment: {{ counter }}</p>
```

```
<button type="button" @click="on_increment_click()">  
    Increment</button>  
</div>  
</template>  
  
<script>  
module.exports = {  
    methods: {  
        on_increment_click() {  
            this.counter++;  
        }  
    },  
    data: function() {  
        return {  
            message: "No current order from you",  
            counter: 0  
        };  
    }  
};  
</script>
```

We return a set of properties named message and counter. We have also defined a method named on_increment_click that increments the counter. The template uses both these properties using mustache syntax. We have also bound the method to the on-click event of the button. If you run the sample application and switch to the My Order tab, you would see the Vue component rendered as shown in Figure 1-16.

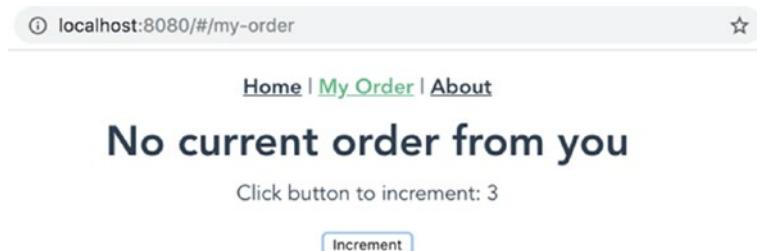


Figure 1-16. The Number Increments upon Clicking the Button

Upon clicking this button, you can see that the variable is incremented and the value in the paragraph is changed automatically. Try and see how easy it is to bind an element in JavaScript with an element in rendered UI if we use a framework like VueJS. You can achieve similar binding even with other frameworks like AngularJS and ReactJS. However, as I mentioned, they have a steep learning curve.

In order to include a component in another one, we can use the template as shown in the following. Here we are getting a food_items array from the store and iterating over it and creating a FoodItem DOM element and passing each item to the food item component for it to use the data inside the item object. You can find the full code for this and how the component is imported in the script element in file Home.vue:

```
<div class="flex-row">
  <FoodItem v-for="item in this.$store.state.food_items"
  :key="item.id" :item="item" />
</div>
```

There are a few other concepts I haven't explained regarding computed properties, executing upon mounting a component, committing changes to store using mutation methods, how the router is configured, and so on. I suggest you go through those by navigating to the following links:

<https://vuejs.org/>

<https://vuex.vuejs.org/> ← Vuex is the framework used in store.js

<https://github.com/championswimmer/vuex-persist> ← Vuex plugin used for local storage

Summary

We have covered a lot of ground in this chapter. If you are new to AWS, you may feel a bit overwhelmed by the sheer amount of information in this chapter. Take your time and try to work with all the examples before you proceed to the next chapters. We have covered essential services that are necessary for us to work with services intended for integrating applications in AWS. We have started the chapter by learning how AWS is organized into sections and understood the problems they solve.

We have then proceeded to learn the Identity and Access Management service (IAM) so that we have sufficient knowledge to give permissions necessary for us to work with any other service in AWS. Using IAM, we can create users and roles with fine-grained access controls to keep our data and application secure.

Next stop is to understand Simple Storage Service (S3) which is a service intended for storing our files on the cloud. We have gone through basic concepts such as buckets and uploading files to the bucket. We have also seen how to run a static web site from S3. We then learned how to use CLI to upload files from the command line without opening the AWS console.

We have then proceeded to learn how to store dynamic data or live data using DynamoDB. It is a NoSQL database service that we can start using within minutes. We have seen how to create tables and insert data into the tables using the AWS console, and then we have seen how to use AWS SDK to insert and read data programmatically. We have also seen examples of how to do that in not just one but three programming languages. We have covered some snippets in JavaScript, Python, and Java. We have also seen how to make use of Cognito identity pools to give programmatic access for the code to interact with AWS services.

In the previous sections, we have been fiddling with running the code locally, but to fully embrace the cloud, we needed to see at least one way to run code on the cloud. We have learned the Lambda service to run code without provisioning any servers. Lambda functions are event driven, and we have seen how we can create them and run them in the AWS console as well as from a web page. We have also seen examples of how a Lambda function can interact with DynamoDB. We have also seen how monitoring of lambda functions can be done using the CloudWatch service.

Finally, we have looked at how our sample application is organized and how to run it and deploy it to the S3 bucket we have created earlier. I have also taken you through basic VueJS component concepts in a flash. It is not enough to understand VueJS fully, but it allows you to start using the code base without too much confusion.

CHAPTER 2

Simple Queue Service

The key to building scalable systems is to have loosely coupled components in our architecture, components that do not have strong dependencies between each other, so that if one component were to die (fail), sleep (not respond) or remain busy (slow to respond) for some reason, the other components in the system continue to work seamlessly as if nothing has happened.

—Jeff Barr¹

The date July 11, 2006, is an important one for the world of cloud computing. Amazon released its first-ever infrastructure web service for the general public.² Many people thought that it must be S3 or EC2 considering the fact they are widely used. However, it was the Amazon Simple Queue Service that was released with not much hype or fanfare. SQS was available from 2004, two years before its general release. Many people wondered as to why an e-commerce web site suddenly started selling a message queue service. On its own, SQS didn't attract much attention from the crowd, but soon it has become a vital piece building loosely coupled scalable systems. It kick-started the whole world of Amazon Web Services. No one has any clue about what is to come next.

¹SQS: Super Queue Service: https://aws.amazon.com/blogs/aws/sqs_super_queue/

²<https://aws.amazon.com/about-aws/whats-new/2006/>

CHAPTER 2 SIMPLE QUEUE SERVICE

Simple Queue Service, or SQS for short, is a powerful idea packaged as a simple product. It allows you to send messages across your application components, without worrying about all the things that can go wrong. Each component that has to send a message to another component or application sends it to Amazon-hosted SQS. All the applications that need to receive the message read from the same queue. If the message is processed successfully, it is deleted from the queue; otherwise, it is retained until it is processed again. This architecture allows you to build reliable and scalable applications without complicating the life of developers. An example might help you understand it better.

Imagine you are setting up a new restaurant. In an ideal world, as soon as a customer orders a dish, it is served to the customer instantly. However, the real world is often different. As soon as we give the order, the waiter hops on to other tables to take a couple of orders. Once he is satisfied that he doesn't have to make too many round trips, he goes to the chef and conveys the orders to him. The chef may have been working on his previous order and can only start preparing yours once he is free and he gets all the ingredients. Once all the preconditions are met, he starts preparing your food and keeps it ready for the waiter to carry it to your table.

If you look carefully, you can find similar examples all around you. You may go to a package service and want to send a gift to your mother on her birthday. As soon as you pay for the delivery and give the package to the courier company, they add it to the outbound package pile, sort it, and group it with all the shipments that have to go to the destinations within a range. The packages are then sent to the target locations and wait for agents for the last mile delivery. Alternatively, you may go to a bank to cash a check. You have to wait in a queue until a teller is available for you to help with that.

I can come up with many such examples, but you see where I'm going. The package or dish may have to sit around for a lot of time to be picked up by the next set of free hands before something can be done about it. There

is a lot of wait time. We can say they are waiting in a queue to be processed. While they are being processed, something may go wrong, and another person may have to come and pick them up.

Most of the cloud applications you interact with are no different. You send an email; it may not go immediately to the recipient. It may be stored in a queue until the next SMTP server is ready to process it. You buy an item on an e-commerce web site; the order will be in a queue until there is a person to pack that item for delivery. Say you are building a video service to which customers can upload their videos and your service generates subtitles. These kinds of queues are necessary for almost all systems that can fail. Any product we build these days is expected to be fault tolerant. Even if some downstream service is down, it must be able to retry the same request after some time.

The systems we are building nowadays have several components that work independently. These components share the responsibility of serving the user, and they communicate with each other as needed. Each component only has code related to fulfilling the core logic, as shown in Figure 2-1.



Figure 2-1. Component Interactions in the Ideal World

Even though we can build these systems, the applications get more complicated as we try to build many features surrounding error handling when the applications we are not interacting with go offline. How do we handle throttling of the service we are interacting with? Can we send these messages in batches? It becomes even more complicated if the service receiving the message needs to work on it for hours before returning a result.

You need to wrap many layers of protection to shield them from all failure scenarios as shown in Figure 2-2. If you have multiple components that run independent of each other and occasionally communicate asynchronously, you need to bloat each of these services with extra functionality that doesn't really add any more value to the customer of your application.

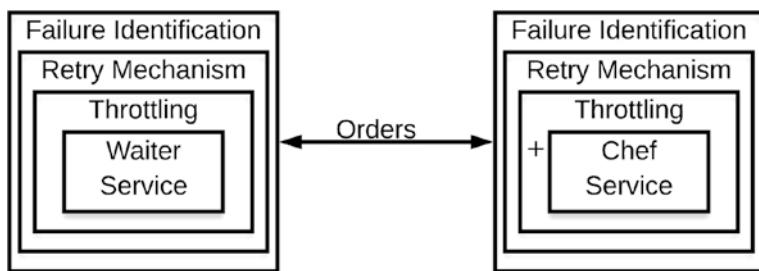


Figure 2-2. *Each Component Is Complex and Bloated*

Integrating these applications quickly becomes a headache as the number grows. Doing all of these at scale complicates it even further. In this chapter, we shall cover Simple Queue Service (SQS) that allows you to focus on the features that add value to your customer by offloading all the added complexity of decoupling services. With SQS in the picture, the applications look like in Figure 2-3.



Figure 2-3. *Simplified Architecture with Queues*

In a perfectly decoupled system, the services should communicate with each other, but failure of one component shouldn't impact others. We can achieve that by introducing services like SQS. The waiter service sends

messages to SQS. SQS keeps track of the messages that were consumed by the chef service, which pulls the messages out the queue and works on them.

The advantage here is that even if chef service is down, the waiter service doesn't have to stop sending messages. The messages will be neatly queued by SQS, and if the consumer service crashes while processing a message, SQS will allow retries once the consumer recovers.

However, as the chef service's regular responsibility is to read the messages and process them, recovery or retry is no different. As developers, we do not have to develop any special measures to ensure that the recovery process goes smoothly. SQS is a managed PAAS service offering, so we do not have to set up or maintain any servers, and Amazon takes care of any queue administration. You have to pay for the number of messages you send and pull. If you maintain the servers on your own, scaling involves setting up more servers and managing them on demand. Amazon scales these services seamlessly depending on how quickly you send messages.

There are two types of charges for using SQS. First is based on the number of requests you make. For a standard queue, it costs you \$0.4 for every 1 million requests. However, the first 1 million requests are free. Each put message call, receive message call, and delete message call are considered separate requests. So for each message we send and receive using a queue, we consume at least three requests. We can save the number of requests by batching the calls. We will get into all those specifics later.

The second type of charge is per GB. The first 1 GB is free. After that, it is 9¢ for every gigabyte of data you read from the queue. However, take this pricing information with a grain of salt. Prices vary for each region, and the prices listed here are for US regions. Another factor is that Amazon constantly reduces the prices for all AWS services. When released in 2006, you paid 1¢ for every hundred messages sent and 20¢ for every gigabyte of data transferred. In 2013, only the first 100,000 requests were free, and the price per request was reduced on a regular basis.

There are a few more theoretical aspects we need to cover before you get a complete grasp of the service, however. Let us first see some practical usage of SQS in a simple scenario so that we become familiar with some aspects.

Big Picture

As you might already know, we are trying to build an online ordering service through which a customer can request for a food to be delivered. Each order goes through a series of steps before it gets delivered to the customer. The flow looks as shown in Figure 2-4.

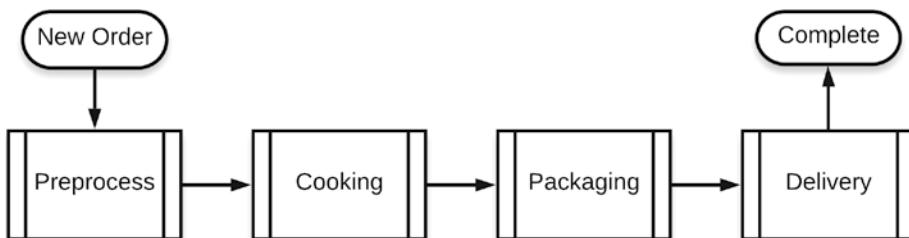


Figure 2-4. Order Delivery Process

Once the order is kicked off, it has to be prepped. All the ingredients have to be gathered and set in an order. Once the preparation/preprocess is over, the cooking process is initiated, and it can take any amount of time. Sometimes the bread gets burned, and it may have to restart from the get-go. Once the cooking is done, it has to be packaged, and a delivery agent has to pick it up and take it for delivery. Once the order is delivered to the customer, it can be marked as complete.

Each phase an order goes through is asynchronous and can take an arbitrary amount of time. Each phase requires the previous phase to be successfully finished. Sometimes, each phase may need to be restarted

because something has gone wrong. However, building this workflow as a series of tightly coupled components is bad for all the reasons we spoke earlier.

Failure in one system causes other systems to fail because of tight coupling. To avoid crashing the components, we need to wrap each of them with fault tolerance, retry, and throttling logics. Each component is going to get more complicated. So instead of complicating each component, we can introduce a queue in front of them, as shown in Figure 2-5, that will take care of resilience.

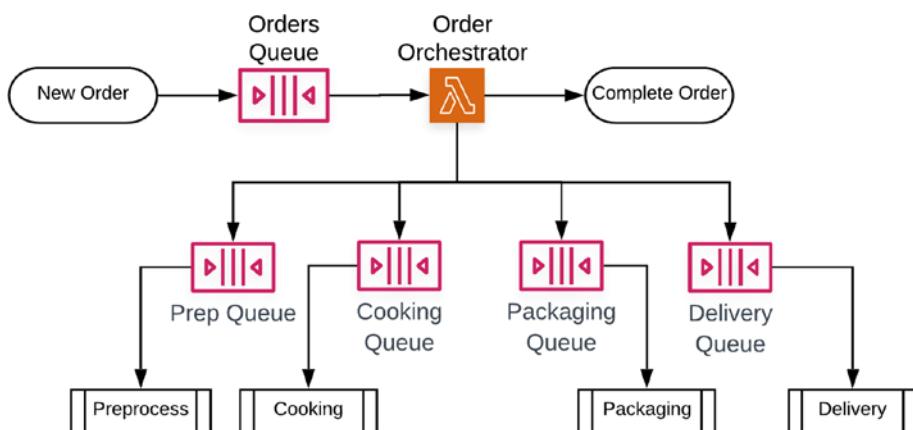


Figure 2-5. Decoupled Services with the Help of Queues

However, there must be an orchestrator component that inserts messages into the correct queues. Every new order that enters the system will be inserted into the queue that is consumed by an orchestrator component and then moved to the prep queue, which is consumed by the preprocess component. Once the preprocess finishes its job, it informs the orchestrator component either by updating a database record or reinserting the message in the orders queue. The order orchestrator picks it up from there to move the order forward in the workflow.

Creating a Queue

As you have seen, we need to first create a queue for us to use it as a communication channel between several components. A queue needs a name and few other attributes which can be configured while creating. The minimum amount of information we have to pass for creating a queue is its name.

The value of the queue name parameter can be alphanumeric characters of maximum length 80. It can also have underscores (_) and hyphens (-) along with alphabets and numbers. There are special types of queues called first-in-first-out (FIFO) queues, which should end with **.fifo**. If there are any invalid characters or if the queue name doesn't end with **.fifo**, the queue creation wizard shows an error prompt while you type the name. We will also go over these special queues in this chapter. There are parameters that are accepted by the create-queue operation.

Each queue can have various preferences that can be configured. For example, we can configure how long the message should stay in the queue before being discarded. We can control how big the messages can be before they get rejected by SQS. These configurations can be passed while creating the queue. Let me walk you through the process of creating it using multiple mechanisms available in AWS. First up is the easiest approach.

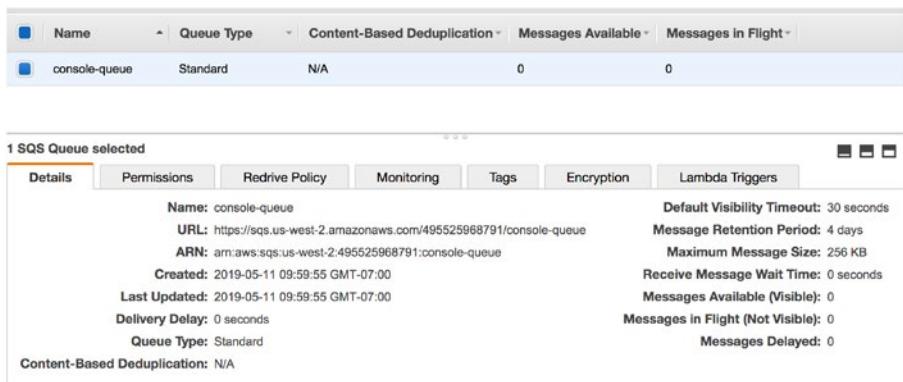
Console Method

Creating an SQS queue is not a repetitive task that you would want to perform again and again. You typically create a queue once and use it as long as your application is live. So you may be better off creating the queue in the AWS console. To create one, you can follow these steps:

1. Open SQS from the AWS Management Console.
2. If you have not created any queues before, you will find a *Get Started Now* button. Click it to start creating a new queue.
3. Click the *Create New Queue* button located on top of the queue list.

Either way, a new queue creation wizard is opened. The first page of this wizard has two choices for you to make. One is the name of the queue, and the other is the type of queue. Click Standard Queue for now. Give a name and scroll down to either finish creating a queue or customize the parameters of the queue. Let us just finish creating the queue and explore what is created by default. The queue will be created in seconds and be ready for you to use instantly.

Select the queue you have just created in the table, that is presenting you the list of queues, by clicking it. The table reduces its height to make way for showing the details as shown in Figure 2-6. The Details tab shows you the simple configurations.



The screenshot shows the AWS Simple Queue Service (SQS) console. At the top, there is a table with one row, showing the queue 'console-queue' with a 'Standard' type and 'N/A' for content-based deduplication. Below this, a modal window titled '1 SQS Queue selected' displays detailed information about the queue. The 'Details' tab is selected, showing the following configuration:

| Name | Queue Type | Content-Based Deduplication | Messages Available | Messages in Flight |
|---------------|------------|-----------------------------|--------------------|--------------------|
| console-queue | Standard | N/A | 0 | 0 |

Details Tab Content:

- Name:** console-queue
- URL:** <https://sqs.us-west-2.amazonaws.com/495525968791/console-queue>
- ARN:** arn:aws:sqs:us-west-2:495525968791:console-queue
- Created:** 2019-05-11 09:59:55 GMT-07:00
- Last Updated:** 2019-05-11 09:59:55 GMT-07:00
- Delivery Delay:** 0 seconds
- Queue Type:** Standard
- Content-Based Deduplication:** N/A
- Default Visibility Timeout:** 30 seconds
- Message Retention Period:** 4 days
- Maximum Message Size:** 256 KB
- Receive Message Wait Time:** 0 seconds
- Messages Available (Visible):** 0
- Messages in Flight (Not Visible):** 0
- Messages Delayed:** 0

Figure 2-6. Queue Details

CHAPTER 2 SIMPLE QUEUE SERVICE

Along with the name, URL, creation time, and whether the type is a standard queue or a FIFO queue, it also shows info about how long the messages will be retained in the queue or the maximum allowed size of the message, how long should the message be hidden from clients before allowing them to pull. The Details tab also shows how many messages are currently in the queue, which is zero as we have just created the queue.

The Permissions tab is to allow cross-account access for the SQS queue. The Redrive Policy is for controlling how many times the messages must be allowed to reenter the queue in case of failure to process. The Monitoring tab shows a time-plotted graph of message arrivals, deletions, and a few other metrics. The Tags tab as you well know is for the purpose of keeping track of costs. The Encryption tab allows us to encrypt the messages on the server using another service called the Key Management Service. The final Lambda Triggers tab is for us to see if we have any Lambda functions created with SQS message as a trigger.

If you want to change any of the parameters, you can do so by opening the configuration wizard by clicking the Queue Actions button and then the Configure Queue menu item from the dropdown menu. This opens up a wizard that displays all the configurable parameters of the queue, as shown in Figure 2-7. If you remember, we chose Quick Create while creating this queue to create this quickly. If we had clicked Configure Queue instead, we would have seen the exact set of configurations to fiddle with.

Configure console-queue

Queue Settings

| | | |
|--|---|---|
| Default Visibility Timeout  <input type="text" value="30"/> | seconds  | Value must be between 0 seconds and 12 hours. |
| Message Retention Period  <input type="text" value="4"/> | days  | Value must be between 1 minute and 14 days. |
| Maximum Message Size  <input type="text" value="256"/> | KB | Value must be between 1 and 256 KB. |
| Delivery Delay  <input type="text" value="0"/> | seconds  | Value must be between 0 seconds and 15 minutes. |
| Receive Message Wait Time  <input type="text" value="0"/> | seconds | Value must be between 0 and 20 seconds. |

Dead Letter Queue Settings

| | |
|---|---------------------------------------|
| Use Redrive Policy  <input type="checkbox"/> | |
| Dead Letter Queue  <input type="text"/> | Value must be an existing queue name. |
| Maximum Receives  <input type="text"/> | Value must be between 1 and 1000. |

Server-Side Encryption (SSE) Settings

| | |
|--|---|
| Use SSE  <input type="checkbox"/> | |
| AWS KMS Customer Master Key (CMK)  <input type="text"/> | |
| Data Key Reuse Period  <input type="text"/> | <input type="text" value="1"/> minutes  This value must be between 1 minute and 24 hours. |

[Cancel](#) [Save Changes](#)

Figure 2-7. Configurations of a Queue

The first section of the configuration wizard is for simple queue settings. Dead Letter Queue is for controlling the redrive policy, and the SSE portion is to keep the messages safe and secure. Make any changes you wish to and save them. I will take you over the advanced configurations in the later sections of this chapter. For now, let us just stick to creating simple queues. The queue we have created can be used immediately by using queue URL.

But before we do that, let us also see how we can use CLI and SDK to create queues. Creating queues in the console is simple, but if we have to repeat the process or create queues at runtime, the console doesn't help much. The other approaches allow us to automate the process of creating queues.

CLI Method

When you are starting small, you can get by with just one queue in a region. However, as your application can grow in number of users and when the users are in different geographic locations, you may have to regionalize your application. In such a case, you have to create the same queue in multiple regions for efficiency and latency's sake. By using the console, we may end up creating queues with different configurations by accident. Using CLI, we can save the configurations as a file and reuse them from the command line as many times as necessary.

Every CLI command takes the same form irrespective of the service. Before you can execute a command, ensure that you have configured a profile with IAM user credentials with access to SQS. You should also configure the region this user uses by default. Set it to the same region as your DynamoDB table or lambda function. I configure it as us-west-2 with profile name `sqs_dev`. If you configure the credentials for the default profile, you can run the command without any profile parameter:

```
$ aws sqs create-queue --queue-name 'cli-queue'
```

The command is familiar if you have run any commands in the first chapter. The profile and region parameters of the command are to control authentication and region. The user has access to do any operation related to SQS. The command **create-queue** takes several parameters. One of

them is **queue-name**. Overall, the create-queue command takes the form shown in the following:

```
create-queue
  --queue-name <value>
  [--attributes <value>]
  [--cli-input-json <value>]
  [--generate-cli-skeleton <value>]
```

The queue configurations can be passed in as attributes. We can either pass a map of parameters as a string or as a JSON file. In the previous section, I have shown various settings applicable for a queue such as retention period; all of them can be passed in as attributes. Look at the following commands, for an example of how you can pass multiple attributes as comma-separated key-value pairs or as a JSON:

```
$ aws sqs create-queue --queue-name 'que' --attributes
"DelaySeconds=9,VisibilityTimeout=7"
Or
$ aws sqs create-queue --queue-name 'que' --attributes
file://create-queue-attributes.json
```

It will be inconvenient to pass more than one or two configurations in the command line. It is more convenient to create a file with the required configurations and pass the file name instead. The JSON file will just be a key-value pair with key names the same as in an attributes string:

```
{
  "DelaySeconds": "9",
  "VisibilityTimeout": "7"
}
```

CHAPTER 2 SIMPLE QUEUE SERVICE

When you execute the command, it returns the queue URL if the creation is successful. The queue URL is in the form “`https://SQS_REGION_ENDPOINT/AWS_ACCOUNT_ID/QUEUE_NAME`.” Based on this URL, we can conclude that the queue names must be unique in a region for an AWS account. You can create a queue with the same name in your prod and test accounts, or you may have the same queue name for queues in different regions.

The `create-queue` command may return various types of errors in case of issues with the input or configurations. If you get an error saying access denied, it just means you are not using the correct profile or the user does not have access to create queues. Go to IAM and give the user sufficient access. Alternatively, while learning, just use the admin user we created and configured as the default user all the time. But it is not a good idea to use the default user with admin access as you may accidentally delete critical production resources. So use your best judgment when configuring the admin user as the default user. It is probably okay for a test account.

Other errors can be that the queue already exists. As mentioned in the preceding text, you cannot have two queues with the same name in a region for an account. Another error that frequently comes up is that there is a typo in attribute names or values.

Another attribute of the `create-queue` command is `cli-input-json`. This is used to store the entire `create-queue` request in a JSON and use it repeatedly. Previously, you have seen how to store attributes as JSON. However, it is also possible for us to store the entire request as a file. We can save the following content as a file and pass it as an argument for this parameter:

```
{  
    "QueueName": "cli-queue-1",  
    "Attributes": {  
        "DelaySeconds": "9",  
        "VisibilityTimeout": "7"  
    }  
}
```

It looks eerily similar to the attributes JSON with just one additional parameter. The queue name is the only additional parameter we are passing in addition to the attributes, so both the JSON files are similar except for that one field. Assuming that we save the preceding content as create-queue.json, we can execute the following commands to create the same queue in different regions:

```
$ aws sqs create-queue --region us-west-2 --cli-input-json  
file://create-queue.json  
$ aws sqs create-queue --region us-west-1 --cli-input-json  
file://create-queue.json  
$ aws sqs create-queue --region us-east-1 --cli-input-json  
file://create-queue.json
```

Executing the preceding commands creates the same queue in three different regions, and the output will be three different queue URLs with the same queue name and account id with a different region endpoint for SQS. In the AWS console, you can switch to a different region to validate if the queues have been created.

The final command line parameter generate-cli-skeleton is for creating an empty create-queue.json template for us to validate the given parameters and return a sample output. This parameter is not unique to the create-queue operation or, for that matter, not just for the sqs command. Many AWS CLI commands support this parameter to simplify CLI usage. This parameter accepts two values, input and output. The value input is default, and it just prints an empty create-queue template JSON. If we pass output as the value for this parameter, it validates the command arguments and returns a sample output if everything checks out. Otherwise, an error is displayed. None of the requests are sent to SQS:

```
$ aws sqs create-queue --queue-name 'cli-queue-5' --generate-  
cli-skeleton output
```

SDK Method

To be honest, you will rarely need to create a queue at runtime. Most of the applications can get by creating queues from the command line or console. However, imagine you are creating an application that allows interaction with multiple services through numerous queues that can be created on demand. You will then need to use SDK if you want to reduce the operational load on the product team. So even if it is a rare occurrence, it may come in handy at times.

Using SDK follows the same approach as it did in Chapter 1. I'm hoping that you have AWS SDK installed for the programming language of your choice. For a NodeJS project, you need to have aws-sdk mentioned in dependencies of package.json file. For a Python project, you need to have boto3 installed through pip. For a Java project, you need to have aws-java-sdk-sqs JAR added to your project dependencies. If you are using maven, make sure that you have the following dependency added to your pom.

xml:

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-sqs</artifactId>
    <version>LATEST</version>
</dependency>
```

I'm going to demonstrate how to do it in JavaScript in Listing 2-1, and if you want to see the code for Java or Python, please refer to the source code. However, it must be easy for you to follow as all those programs take a similar approach.

Listing 2-1. javascript-sources/chapter-02/demo-01/create-queue.js

```
const AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.SharedIniFileCredentials
({profile: 'sqsh_dev'});

let sqs = new AWS.SQS();

var params = {
    QueueName: 'programmatic-queue-1',
    Attributes: {
        'DelaySeconds': '5'
    }
};

sqs.createQueue(params, function(err, data) {
    if (err)
        console.log(err, err.stack);
    else
        console.log(data);
});
```

If you have executed code samples in Chapter 1, this should be very familiar. First, we get credentials either using Cognito or from local configuration (configuration done using the `aws configure` command). We can use local configuration for server-side code and Cognito for client-side code. Once credentials have been obtained, set them to the AWS config and create an SQS client by `new AWS.SQS()`. This client can be used to invoke the API necessary to create a queue.

The `createQueue` method accepts a JavaScript object encapsulating queue name and attributes. This is the same as the JSON file used from the command line while creating the queue. Along with parameters, it also

CHAPTER 2 SIMPLE QUEUE SERVICE

accepts a callback function that gets called once the operation is either successful or failure. The successful response returns data with queue URL, whereas any issues with the input or access return an err object. Running the preceding program returns this output:

```
{  
    ResponseMetadata: {  
        RequestId: '9b5dbcd4-f831-510f-a09e-e985be0917f4'  
    },  
    QueueUrl: 'https://sns.us-west-2.amazonaws.  
                com/495525968791/programmatic-queue-1'  
}
```

If we look at Java code as shown in Listing 2-2 to achieve the same, we create a HashMap of the attributes and use it to create a CreateQueueRequest object. We then invoke the createQueue method of the SQS client and pass the request object we constructed. This method call invokes SQS to create a new queue and returns the result if it is successful. In case of any issue, an exception is thrown.

Listing 2-2. java-sources/chapter-2/demo-01/CreateQueue.java

```
import com.amazonaws.auth.profile.*;  
import com.amazonaws.services.sqs.*;  
import com.amazonaws.services.sqs.model.*;  
  
import java.util.*;  
  
public class CreateQueue {  
    public static void main(String... args) {  
        ProfileCredentialsProvider credProvider = new  
        ProfileCredentialsProvider("sns_dev");
```

```

AmazonSQS sqs = AmazonSQSClientBuilder.standard()
        .withCredentials(credProvider)
        .build();

Map<String, String> attributes = new HashMap<> ();
attributes.put("DelaySeconds", "5");

CreateQueueRequest request = new CreateQueueRequest()
        .withQueueName("programmatic-queue-3")
        .withAttributes(attributes);

CreateQueueResult result = sqs.createQueue(request);
System.out.println(result.getQueueUrl());
}

}

```

Whichever approach you use, the behavior and end result are the same. You pick the approach or language suitable for your requirement.

Send Messages to Queues

So far, we have seen how to create a queue in a region for our consumption. By consumption, I mean using it as a message pipe between two separate components. It is a two-step process; a producer creates a message and sends it to a queue and proceeds to its next job without waiting for the message to be consumed. A consumer regularly polls the same queue, and as soon as it sees a new message, it pulls the message and processes it. If the message processing is successful, it deletes the message in the queue; otherwise, the message is retained in the queue for consumption by the same consumer again or another consumer.

In an elastic world of cloud computing, both producers and consumers can be multi-node clusters. A consumer can be a chef service running as a process on one or more computers waiting for UI to send a message or a

Lambda function ready to process as soon as it sees a new message enter the queue. If UI is inserting messages to the queue when a customer clicks the Order button, it can be considered as the producer. In this section, we will focus on the producer; and in the next section, we will learn how to create consumers.

Console Method

Whenever you want to quickly send a message to invoke a consumer and test its behavior, it is very convenient to use the AWS Management Console for that. Open the SQS console and select a queue you want to send a message to. Open the send message wizard by navigating to the option **Queue Actions ➤ Send a Message**. The wizard should look as shown in Figure 2-8. Type any message you want to send to the queue.

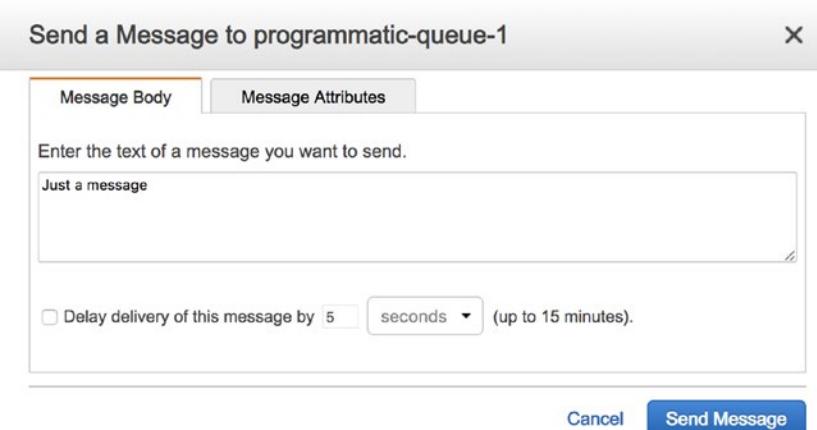


Figure 2-8. Sending Message to a Queue

Along with the message to be sent, we can configure a few other parameters. The most important configuration we can do on this page is the initial delay. The message will not be available for consumers to pull until after the delay period.

Before the delay period runs out, the Details tab of the queue will show that one message is delayed as shown in Figure 2-9. After the delay period, the message goes into an available bucket. Until then, polling for new messages doesn't return this message. The same information should also be shown in the queue list if the Messages Delayed column is visible. You can configure the visible columns by clicking the gear icon above the queue list.

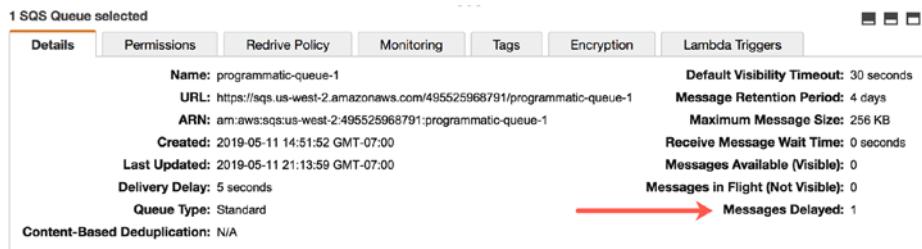


Figure 2-9. Delayed Message

You can add multiple attributes as shown in Figure 2-10. You can also add up to ten optional attributes which are nothing but structured metadata. You can use these attributes to decide whether to process a specific message or not. Occasionally, the message processing can be costly due to the size of the message. Using attributes, we can decide to fail early and stop processing the message further.

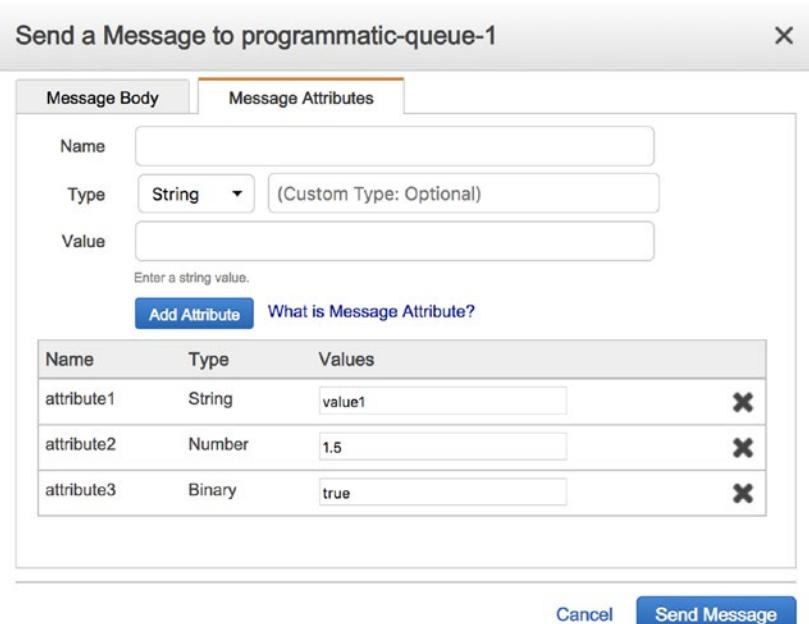


Figure 2-10. Optional Attributes

CLI Method

We can use CLI to send messages as with any other operation that uses SDK. We may need to do so for debugging some issues or some manual operation works to fix or reproduce an issue. There may be times where you can't access the console. To send messages, CLI is a quick and dirty way to do it. The command's syntax is as shown in the following:

```
send-message
--queue-url <value>
--message-body <value>
[--delay-seconds <value>]
[--message-attributes <value>]
[--message-deduplication-id <value>]
```

```
[--message-group-id <value>]
[--cli-input-json <value>]
[--generate-cli-skeleton <value>]
```

As with the previous command, the `generate-cli-skeleton` option, by default, prints out a template you can edit and save as a file to be used along with the option `cli-input-json`. We can also use the `generate-cli-skeleton` option to validate the command parameters by providing its value as output.

The options `message-deduplication-id` and `message-group-id` are only applicable for FIFO queues, so I will explain their usage later. The minimum set of options required to send a message is the queue URL and message body. The queue URL is the value returned by the `create-queue` command, or if you want to know the URL later, you can get it from the console in the Details tab when a queue is selected. Here is a sample command:

```
$ aws sqs send-message --message-body 'this is a test message'
--queue-url https://sqs.us-west-2.amazonaws.com/495525968791/
programmatic-queue-1 --delay-seconds 70
```

This command sends the message to the queue `programmatic-queue-1`, and the message will not be visible until after 70 seconds since running the command. The other option is message attributes, which are optional and can have values of different data types. You need to pass the name of the attribute, the data type, and the value. It would be easier to save it as a file and use it in the command line. It can be done with the help of the `cli-input-json` option:

```
{
    "QueueUrl": "https://sqs.us-west-2.amazonaws.
                  com/495525968791/programmatic-queue-1",
    "MessageBody": "test message body",
    "DelaySeconds": 9,
```

```
"MessageAttributes": {  
    "attribute1": {  
        "DataType": "String",  
        "StringValue": "value1"  
    },  
    "attribute2": {  
        "DataType": "Binary",  
        "BinaryValue": "value2"  
    }  
}
```

If you need a reminder on how to use the file, I have given that in the following assuming that the name of the file is send-message.json. You need to prefix the file name with file://:

```
$ aws sqs send-message --cli-input-json file://send-message.json
```

Executing the preceding command returns the message id if everything is fine with the configurations and the syntax in the file. We would rarely need to use the message id as producers really don't worry about what happens with the message once it is successfully sent.

SDK Method

Now let us go to the meaty part of this whole segment, sending messages. Sending messages through an SDK is the most widely used method. To do it, we have to invoke the sendMessage method on the SQS client as demonstrated in Listing 2-3.

Listing 2-3. javascript-sources/chapter-2/demo-01/send-message.js

```
const AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.SharedIniFileCredentials
({profile: 'sqS_dev'});

let sqs = new AWS.SQS();

var params = {
    "QueueUrl": "https://sqS.us-west-2.amazonaws.com/
        495525968791/programmatic-queue-1",
    "MessageBody": "test message body",
    "DelaySeconds": 9,
    "MessageAttributes": {
        "attrib1": {
            "DataType": "String",
            "StringValue": "value1"
        },
        "attrib2": {
            "DataType": "Number",
            "StringValue": "10.75"
        }
    }
};

sqs.sendMessage(params, function(err, data) {
    if (err)
        console.log(err, err.stack);
    else
        console.log(data);
});
```

This JSON is exactly the same as the one we have used as input while running the command in terminal. Usually, the queue URL is read from a configuration file and constructs the params object. Run this program to send a message to the given queue. You can open the queue in the console to see that the message count has increased.

Pull Message from a Queue

Sending messages to a queue is just half the war. For the message to be useful, a consumer has to pull it from the queue and process it. We can either pull each message individually or get a bunch of messages in a batch. It is efficient to pull multiple messages in a batch.

Console Method

For debugging purposes, we can pull messages in the console by going to **Queue Actions > View/Delete Messages**. This opens a pop-up wizard that has a button for you to click to start polling. Click it to see the messages currently waiting in queue, as shown in Figure 2-11.

The screenshot shows a browser-based interface titled "View/Delete Messages in programmatic-queue-1". At the top, there are two input fields: "View up to: 10 messages" and "Poll queue for: 30 seconds". To the right of these are two buttons: "Polling for Messages..." and "Stop Now". Below the inputs is a note: "Polling for new messages once every 2 seconds." A progress bar at the bottom indicates "4%". The main area is a table with columns: Delete, Body, Size, Sent, and Receive Count. The table contains four rows of data, each with a "More Details" link. The data is as follows:

| Delete | Body | Size | Sent | Receive Count | |
|--------------------------|-------------------|----------|-------------------------------|---------------|------------------------------|
| <input type="checkbox"/> | test message body | 17 bytes | 2019-05-12 01:08:01 GMT-07:00 | 6 | More Details |
| <input type="checkbox"/> | test message | 12 bytes | 2019-05-11 23:47:30 GMT-07:00 | 12 | More Details |
| <input type="checkbox"/> | test message body | 17 bytes | 2019-05-12 00:41:59 GMT-07:00 | 7 | More Details |
| <input type="checkbox"/> | test message body | 17 bytes | 2019-05-12 00:10:12 GMT-07:00 | 10 | More Details |

At the bottom, a note says "Polling the queue at 2.8 receives/second. Stopping in 28.6 seconds. Messages shown above are currently hidden from other consumers." There are "Close" and "Delete Messages" buttons at the bottom right.

Figure 2-11. Messages Polled from the Console

There are several aspects I want to bring to your notice. The default configuration for polling is to view up to ten messages and poll for 30 seconds. It will keep polling either until it receives ten messages or for 30 seconds. You can adjust these settings before starting to poll for the messages. As the message below the table indicates, the wizard is performing 2.8 receives for each second. It sends a request for a new message, and SQS returns immediately whether any new message is available or not. It is a short poll.

The **Receive Count** column in the table shows how many times the message has been pulled by any consumer. Even polling from display counts as a receive. You can verify that by polling for a message more than once. Each poll increases the count by one, assuming that the same message has been received. In a standard queue, there is no guarantee of ordering of messages. So each time you poll, a different set of messages may show up.

The final column in the table with hyperlink More Details is for you to see additional metadata associated with each message. It gives details about who sent the message, when it has been last received, and what are the attributes attached to the message.

The view messages wizard or message details wizard is read-only. You cannot modify the message or its attributes here. The only modification you can do is to delete the message from the queue so that it is not available for other consumers anymore. Once deleted, subsequent polls won't receive this message; however, keep in mind that the message might have already been received by another consumer after it became available once the console polling operation is done. SQS cannot take back the message that has already been sent to another consumer even if you delete it.

While the polling operation is in progress, the wizard shows a message "Messages shown above are currently hidden from other consumers" underneath the progress bar. In other words, even if there are other components polling for messages, they won't see these messages until after the view messages wizard finishes the polling operation; it effectively

locks them from further consumption. As soon as the polling operation is complete, the message changes to “Messages shown above are now available to other consumers.” However, other configurations attached to a message, such as visibility timeout, may still impact whether the message is hidden or not. More on that later.

CLI Method

If you do not have access to the console to pull messages, you can use the receive-message command to fetch messages. By default, it is a short poll. So whether the queue has any messages or not, the response will be instantaneous. If you want to wait until at least one message is available to return, you can use the option `wait-time-seconds`. The following is the syntax for the command:

```
receive-message
  --queue-url <value>
  [--attribute-names <value>]
  [--message-attribute-names <value>]
  [--max-number-of-messages <value>]
  [--visibility-timeout <value>]
  [--wait-time-seconds <value>]
  [--receive-request-attempt-id <value>]
  [--cli-input-json <value>]
  [--generate-cli-skeleton <value>]
```

The option `queue-url` is the minimum information you have to provide to the `receive-message` command. The options `cli-input-json` and `generate-cli-skeleton` work exactly the same as for other commands, so I don’t want to go into the details of these two options.

The options `attribute-names` and `message-attribute-names` return additional metadata attached to the message. However, there is a subtle difference between them. The option `attribute-names` returns some

mandatory metadata automatically managed by SQS such as receive count, sender id, and sender timestamp. The option message-attribute-names allows you to return attributes we added while inserting the message to the queue. For both the attributes, you can either give exact attribute names or just provide all to get all the attributes.

The option max-number-of-messages caps the number of messages returned to the number provided as the value for this option. However, the number of messages returned may be less than the given value. You can control how long the message is hidden from further polling by using the option visibility-timeout:

```
$ aws sqs receive-message --visibility-timeout 3600  
--message-attribute-names attrib1 attrib2 --attribute-  
names All --queue-url https://sqs.us-west-2.amazonaws.  
com/495525968791/programmatic-queue-1
```

The above command, for example, returns one message available in queue. All attributes managed by SQS are returned, along with the custom message attributes named attrib1 and attrib2. Because the visibility timeout parameter has the value 3600, the message will be hidden from further processing for the next 1 hour. The following output is returned when I execute the preceding command. I have truncated ReceiptHandle, MD5OfBody, and MD5OfMessageAttributes:

```
{  
  "Messages": [  
    {  
      "Body": "this is with 10 attributes",  
      "ReceiptHandle": "AQEB...",  
      "MD5OfBody": "2bf0...",  
      "MD5OfMessageAttributes": "2b95...",  
      "MessageId": "129c94d6-f7d9-444e-b843-7dadf202ec34",
```

```

    "Attributes": {
        "ApproximateFirstReceiveTimestamp":
        "1557684219300",
        "SenderId": "AIDAIKWP3W3DCQMVZ4724",
        "ApproximateReceiveCount": "9",
        "SentTimestamp": "1557684219300"
    },
    "MessageAttributes": {
        "attrib2": {
            "DataType": "Binary",
            "BinaryValue": "dmFsdWUy"
        },
        "attrib1": {
            "DataType": "String",
            "StringValue": "value1"
        }
    }
}
]
}

```

The option receive-request-attempt-id is only applicable for FIFO queues and is used for deduplication of receive message calls. Each receive-request-attempt-id is valid for 5 minutes and can be used to get the same set of messages back.

SDK Method

Once you have understood the relationship between CLI and SDK, it is very easy to translate any CLI command input to SDK parameters as long as we are using JavaScript SDK. Other programming languages may be slightly different in terms of syntax, but the approach would be pretty

much the same. However, I'm going to slightly take a different approach here to write some clean code.

AWS JavaScript SDK has one benefit over the other languages. Along with executing a specific service call such as `receiveMessage` or `sendMessage`, it can also return a promise. We can do so by calling the `promise` method on any service call like `sqs.sendMessage(params).promise()`. This promise helps us write cleaner code to manage asynchronous flows instead of using callbacks and avoid callback hell. The following code snippet makes use of promise and implements separate flows for success and error scenarios. The code in Listing 2-4 helps us fetch a message from `programmatic-queue-1` and prints it out:

Listing 2-4. `javascript-sources/chapter-1/demo-01/read-messages.js`

```
let params = {  
    "QueueUrl": QUEUE_URL,  
    "AttributeNames": ["All"],  
    "MessageAttributeNames": ["attrib1", "attrib2"]  
};  
  
let promise = sqs.receiveMessage(params).promise();  
  
processMessage = function(data) {  
    console.log(JSON.stringify(data));  
};  
  
processError = function(error) {  
    console.log(error, error.stack);  
};  
  
promise.then(data => processMessage(data))  
    .catch(error => processError(error));
```

The previous example returns a promise that's fulfilled with a data object or rejected with an error object. If you use promises, a single callback method is not responsible for detecting errors. Instead, the appropriate callback is called based on the success or failure of the request. The logic of each flow is neatly segregated into its own method making it less cumbersome when we have to write much more complicated consumers.

One aspect I have mentioned earlier is that the consumers process each message pulled from the queue and if the processing is successful, they make another service call to delete the message from the queue. Our previous examples don't involve this step. To delete a message, we need to invoke `deleteMessage` and pass the receipt handle we received from the `receiveMessage` service call. The following code snippet in Listing 2-5 demonstrates how to do that.

Listing 2-5. Pulling messages from the queue and processing them

```
let receiveMessageRequest = {  
    QueueUrl: QUEUE_URL,  
    MaxNumberOfMessages: 1  
};  
  
deleteMessage = function(handle) {  
    let deletePromise = sqs.deleteMessage({  
        QueueUrl: QUEUE_URL,  
        ReceiptHandle: handle  
    }).promise();  
    deletePromise.then(data => console('Message successfully  
removed'))  
        .catch(error => console.log(error, error.stack))  
};
```

```
processData = function(data) {  
    let messages = data.Messages;  
    if (messages && messages.size > 0) {  
        let handle = messages[0].ReceiptHandle;  
        console.log('Message Received');  
        deleteMessage(handle);  
    } else {  
        console.log('No Messages');  
    }  
};  
  
let receivePromise = sqs.receiveMessage(receiveMessageRequest).  
promise();  
receivePromise.then(data => processData(data))  
.catch(error => console.log(error, error.stack));
```

The preceding code snippet calls `receiveMessage` API, and the data returned by it is passed on to the function `processData`. The function `processData` checks if there is at least one message and, if so, just prints out the message and invokes `deleteMessage` API with the receipt handle from the message object.

You can execute this code as many times as you want, and each time it prints out the message and deletes it from the queue. What we have just done here is that we created two components of an application that communicate with each other asynchronously with the help of SQS. The same principles we learned here can be expanded to building complex applications that can scale easily.

Long Polling

Amazon SQS maintains each of your message in multiple servers, and when we make a call to receive a message, only a subset of servers are queried based on weighted random distribution to determine if any messages are available for a response. Only the messages available on sampled servers are returned, and the response will be sent back immediately. This is the default behavior of SQS and is called short polling. This may result in false, empty responses occasionally, even when there are messages available in the queue.

As you may remember from the earlier section of this chapter, we are charged per request. Typically, we poll the queues on a scheduled basis so even if we receive a false, empty response in one request, there is a high chance that the subsequent request will return a non-empty response. This may essentially increase the number of requests we make to SQS, and our servers also have to increase network throughput and CPU cycles to accommodate for a higher number of requests we make to SQS in a background process. Even in the cases where messages are pushed to the queue, the queue may be empty when the receive message request is made, but as soon as we responded with an empty response, a new message will be pushed to the queue. The end result is that the overall cost is higher than it needs to be.

In November 2012, around 8 years after launch, Amazon released a feature known as long polling. With this feature, instead of returning immediately, it waits until given timeout or at least until one message is available to return. In this approach, all servers are queried to ensure that there are no more messages to return before waiting for new message arrivals. We can configure the long polling timeout to be anywhere between 0 and 20 seconds. We can set this property for a queue to ensure that each receive message request waits for a given number of requests or it can be overridden at the receive request level.

When using CLI for receive message, the parameter `--wait-time-seconds` is used to control how long the polling should be, the duration (in seconds) for which the call waits for a message to arrive in the queue before returning. If a message is available, the call returns sooner than the wait time seconds. If no messages are available and the wait time expires, the call returns successfully with an empty list of messages.

This can be configured globally for a queue when creating/configuring it, setting a non-zero value for the attribute `ReceiveMessageWaitTimeSeconds`. If you are creating the queue through the console, you can provide this value to the attribute with the same name in the queue settings as shown in Figure 2-12. By configuring it at the queue level, we don't have to explicitly set the `--wait-time-seconds` for each receive message request, unless we want to override the value.



The screenshot shows a configuration interface for a Lambda function. On the left, there's a section labeled "Receive Message Wait Time" with a small info icon. To its right is a text input field containing the value "0". To the right of the input field is the unit "seconds". Further to the right, a note says "Value must be between 0 and 20 seconds."

Figure 2-12. Long Polling Configuration

Life Cycle of a Message

Every message we send to a queue goes through a series of stages before it gets processed and removed from the queue eventually. The message may not be ready for consumption by the clients immediately. It depends on the queue visibility timeout or individual message timeout configurations. Each attempt at pulling the message from the queue is kept track of, and SQS will temporarily hide the message so that it is not processed by multiple clients simultaneously. So a lot of things affect what happens once a message enters a queue. The entire life cycle is shown in Figure 2-13.

CHAPTER 2 SIMPLE QUEUE SERVICE

While creating the queue, you may have seen a configuration named delay seconds. The same property can also be set when sending a message to the queue. This configuration is to control how long should the message be hidden before it is visible for consumers to pull. Once the delay seconds are complete, the message becomes visible, and any clients that pull for new messages may get this message. If we configure different delay seconds for the queue and message, the configuration for the message wins. So if we create a queue with delay seconds of 5 and send a message with delay seconds of 10, the message will be hidden for 10 seconds.

Once the message is available for consumption, the next request may send this message, because there can be many messages in an available state and there is no guarantee of the order in which they will be picked unless the queue is a FIFO type. However, if the message is not picked until the retention period runs out, the message may get removed without being received even once. Essentially, it is a Time to Live (TTL) configuration for each message that enters a queue. The retention period is a configuration we set on a queue while creating it.

Once a client pulls the message, it starts processing it, and the message is considered to be in flight. The process may be to display the ingredients for a cook to prepare a meal or trigger a robotic arm to start cutting the vegetables that go into the preparation. The process can be anything, and it can take any amount of time as long as it is between 0 seconds and 12 hours. The visibility timeout begins as soon as the consumer receives the message. We can configure visibility timeout for the queue which is applied to all the messages that enter a queue, or we can change this configuration upon each pull. All these configurations control when the message enters and leaves a state as shown in Figure 2-13.

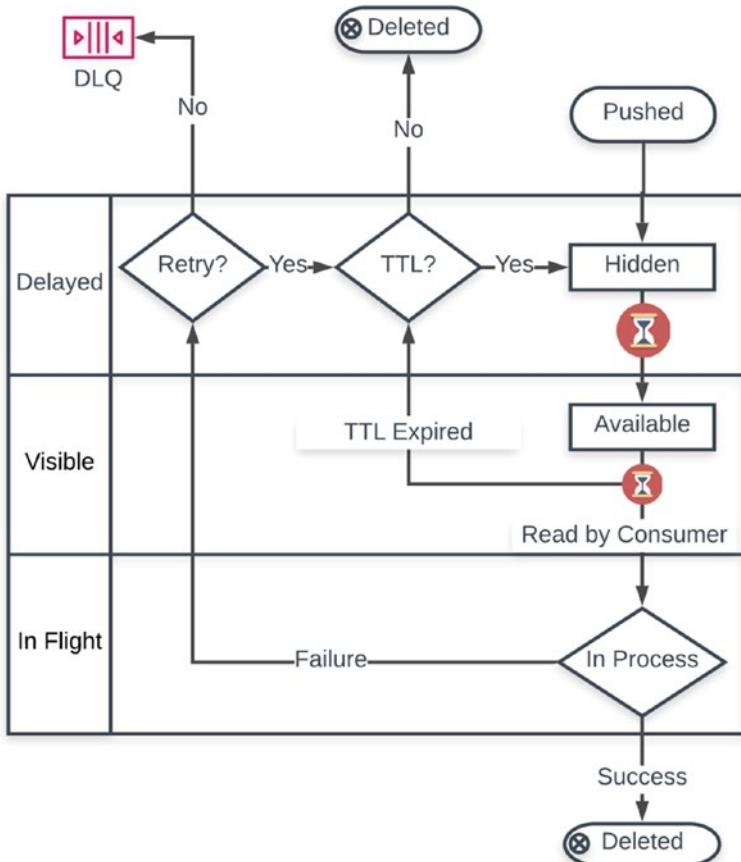


Figure 2-13. Life Cycle of an SQS Message

While a consumer is working with the message, there are two likely outcomes. One, the meal may be prepared up to the satisfaction of the cook, which means the processing has been succeeded. Two, it may be overcooked making the attempt a failed one. In case of successful processing, there is no reason for the message to be in queue; no more chefs have to pick up that order again to prepare. So the message gets deleted from the queue. If there are issues with processing the message,

it will reenter the queue, and another chef may be ready to redo the same order.

Insanity is doing the same thing over and over again and expecting different results.

—Anonymous

This is a famous quote often attributed to Albert Einstein, even though there are no conclusive proofs to support the theory of its origin. However, in the world of cloud computing, we don't want to quit as soon as we fail to process a message. Sometimes issues may be caused due to network interruptions or may be due to a power failure in the server room. In such cases, a retry after some time may fix the issue. However, we cannot end up repeating the same failure message forever. Maybe there is something wrong with the message. Retrying the same message over and over again may end up wasting our resources. We need to strike a balance and handle repeat failures differently from the other messages in the queue.

Each time the message has been received, SQS increments the receive count. Each time the receive count is incremented, it is most likely caused by a failure, so the receive count is a good indicator of how many times processing a specific message failed. Once it reaches a certain number, there is no point in trying it further. In such cases, AWS allows you to redirect the messages to a separate queue. These are called dead letter queues. We will go over those configurations later.

Consumption Approaches

So far, you have seen little snippets of how to consume one or more messages. However, what if we want a consumer that runs all the time instead of when we explicitly call it? We can create a background process that runs regularly at a scheduled interval. We are going to see how to do

that through an application that runs both serverless and on a server. We are going to create workers that don't rely on other pieces of code to invoke them.

Let us first create a queue to hold orders for consumption by workers. We will simply send a message to the queue with an order id. As soon as our web application receives a response after clicking the Order button, we send the order id as the message to a queue named `orders_queue`. It is a standard queue with a visibility timeout of 30 minutes and a retention period of 12 hours, because orders in a restaurant don't really stay active after a couple of hours. If we don't deliver the item quickly, the customer is more likely to cancel the order and leave, so there is no need to retain the message for longer than a couple of hours. The same is the case with an online store like ours that allows home delivery. It is very rare that a customer is willing to wait for 12 hours after he orders. The queue configuration is as shown in Figure 2-14.

| Queue Attributes | | | |
|-----------------------------------|---------------------------------|---------|---|
| Default Visibility Timeout | <input type="text" value="5"/> | minutes | Value must be between 0 seconds and 12 hours. |
| Message Retention Period | <input type="text" value="12"/> | hours | Value must be between 1 minute and 14 days. |
| Maximum Message Size | <input type="text" value="1"/> | KB | Value must be between 1 and 256 KB. |
| Delivery Delay | <input type="text" value="0"/> | seconds | Value must be between 0 seconds and 15 minutes. |
| Receive Message Wait Time | <input type="text" value="20"/> | seconds | Value must be between 0 and 20 seconds. |

Figure 2-14. Orders Queue Attributes

I'm making use of long polling for all the receive message calls by creating the queue with the receive wait time parameter with a value of 20 seconds. I have also configured the maximum message size as we are not planning to send any more than the order id, which will be later queried from DynamoDB. It is not the only way. As we want to get other information like food id, we may send that as well in the queue message

body. This eliminates any need for making an extra call to the orders table to get the food id associated with the given order.

Before we can start consuming messages from this queue, we need a producer. We already have a mechanism to insert data into DynamoDB from a lambda function. The lambda function returns the user_id and order_time which can be placed into the queue from JavaScript running on the browser, or we can ensure that the lambda function inserts the record in the orders queue once it inserted the data in DynamoDB. Either way, the end result is that orders_queue will get a message to kick-start whole orchestration.

In this particular use case, I have implemented the change in the Lambda function. You can find the modified lambda function handler in the source file `javascript-sources/chapter-2/demo-02/lambda-function-handlers/save_order_lambda_handler.js`.

Lambda SQS Worker

Consuming the messages in `orders_queue` using a lambda function is a simple approach. We create a lambda function and add SQS as a trigger for the lambda function to be invoked. We can control how many messages are batched together in each lambda function invocation. With the help of the lambda function, we don't have to have any running servers. Especially if the frequency of the SQS messages is very low, Lambda functions give you a tremendous advantage in terms of cost.

We have already covered what the lambda functions are and how to invoke and test them. So we are not going to repeat those basics again. We will see how to create them and how to generate the test SQS message and so on, to run the lambda with fake data to test. We don't have to start from scratch either when creating the lambda function. AWS gives a blueprint we can use to quickly create an SQS stream worker using lambda functions. Let us first understand the functionality we are going to develop. It is shown in Figure 2-15.

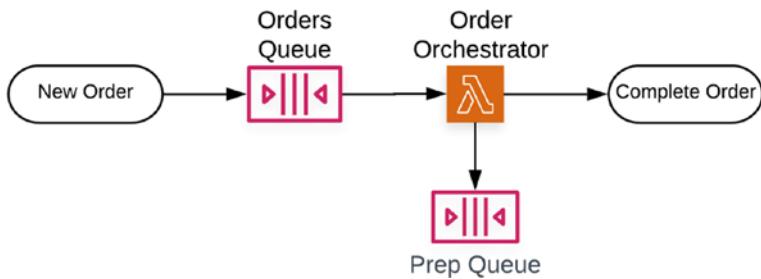


Figure 2-15. *Lambda SQS Worker*

In a restaurant, whenever an order is placed, it will be put in a queue; and as soon as a chef is ready, he picks up the order and gets the details of what item he is supposed to cook and minute instructions of how much cheese the customer likes and if he wants to avoid nuts in his meal and so on. An assistant will see the ingredients on a screen and will gather each of them and mark the task ingredients collection as completed once he collects all the ingredients.

In this section, we will be building a Lambda worker of an SQS stream which will read from the orders queue, and if the state of the order is NEW, it will be placed in a preparation queue. It is the responsibility of a preparation service to read the queue and do whatever it has to do, but we don't have to worry about that at this point in time. For now, we will just focus on building the orchestrator.

Creating the Lambda Function

We don't have to create the lambda for polling SQS from scratch. Lambda provides several blueprints that we can use. In the previous chapter, we have created one from scratch. Blueprints are just getting started convenience. They contain sample code and configuration presets for common use cases. To create one, open the Lambda service and click the Create Function button. This opens a wizard with three options. In earlier instance, we have selected the option Author from scratch. Now select the

CHAPTER 2 SIMPLE QUEUE SERVICE

option Use a blueprint and search for SQS. You should see a search result named `sqs-poller` as shown in Figure 2-16.

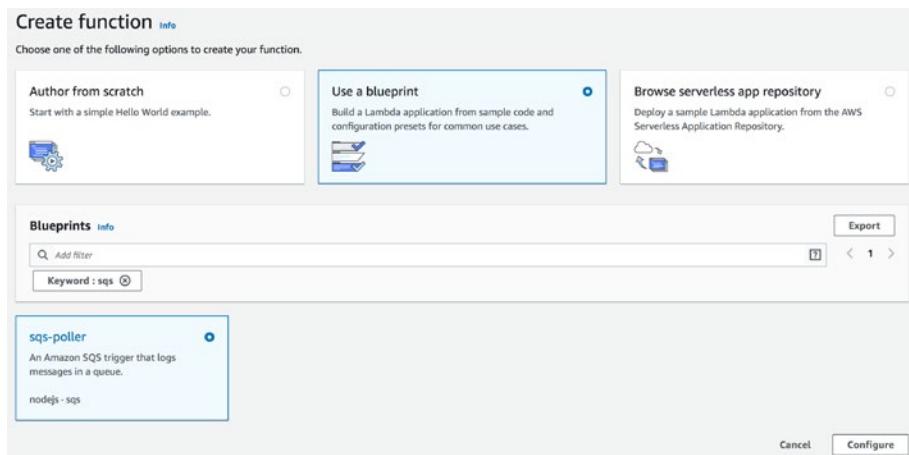


Figure 2-16. SQS Poller Blueprint

Select `sqs-poller` and click Configure to start creating the lambda function. Provide the name as `order_orchestrator` in the basic information section of the Create Function page. Let us give a new name for the role of an orchestrator. The orchestrator is supposed to have access to multiple SQS queues and DynamoDB tables. There must already be some policy template named Amazon SQS poller permissions attached to it. As you already know, the policy is nothing but a set of permissions. As the orchestrator needs to read DynamoDB tables, you already know how to add access using the IAM service if you have read Chapter 1.

If you scroll further down, you will see a section named SQS trigger. In here, we can provide the ARN of the queue we are supposed to listen to. Amazon Resource Names or ARNs uniquely identify AWS resources. We can use ARN when we need to specify a resource unambiguously across all of AWS, such as in Lambda triggers, IAM policies, Amazon Relational Database Service (Amazon RDS) tags, and API calls. You can get ARN for

any resource from the AWS console. For an SQS queue, open SQS from the console and select any queue. A window pops up in the bottom of the screen with tabs like Details, Permissions, and so on. You have already seen the Details tab in Figure 2-6. The Details tab will contain ARN for the queue. Copy the ARN and paste it in the lambda function's SQS trigger. Each ARN consists of service name(sqs, dynamodb, etc.), region(us-west-2, us-east-1, etc.), AWS account id, and the name of the resource. The ARN for my queue looks like this:

arn:aws:sqs:us-west-2:495525968791:orders_queue

In the same tile where you have provided the ARN for your queue, you have to provide the batch size as well. This controls how many messages are handled by each invocation. The maximum size is 10. Change it to 1 for now. We don't need batching here. Scrolling further down brings up sample code. You will not be able to modify the code in this page, but the code is pretty easy to understand. It should look as shown in the following, provided that AWS didn't change the blueprint:

```
console.log('Loading function');

exports.handler = async (event) => {
    //console.log('Received event:', JSON.stringify(event,
    null, 2));
    for (const { messageId, body } of event.Records) {
        console.log('SQS message %s: %j', messageId, body);
    }
    return `Successfully processed ${event.Records.length}
messages.`;
};
```

Whenever this lambda function is invoked automatically because of SQS trigger, the event object contains a Records property. The preceding code is iterating over each message and printing it to the console. We are going to add our orchestration logic here. But first, click the Create Function button to create the lambda function. This will create the role orchestrator and the lambda function `order_orchestrator`. Go to the IAM service and provide the DynamoDB access to orchestrator role. We are not done with IAM yet. The default access configured by the `sqs-poller` blueprint only gives read-only access to queues, but we need to be able to push data to queues, hence give full access to SQS as well.

Once you successfully add DynamoDB and SQS access to the role, you should see them in the Designer tile of the Lambda function. It should show SQS in the triggers and SQS, CloudWatch, and DynamoDB on the access side of the Lambda designer.

We can also create the same Lambda function without using a blueprint. All we have to do is create an empty Lambda function and, in the Designer tile, select SQS from the list of available triggers. You need to enter ARN of the queue we want to listen to and batch size just like you have done earlier. You need to make sure that the role we create has access to SQS, CloudWatch, and DynamoDB at the least. The blueprint simplifies some of the boilerplate tasks. We can add multiple queues as triggers to the same lambda function. However, the logic we write in lambda function is usually written in such a way that it expects specific parameters in the message body. If we don't have such restrictions or expectation, a single lambda function can handle any number of queues.

Invoking the Lambda

Once the lambda function is created, you can test it by configuring a test event and running a test, or you can even send a message to the queue so that the lambda function is triggered automatically. Let us test the integration first. Open SQS and select `orders_queue`. You can send a

message by choosing **Queue Actions ▶ Send a Message**. This opens up a wizard that allows you to send a message. Enter the following JSON in the body and click Send Message:

```
{  
  "user_id": "ce191ec0-6ecb-11e9-83a6-cd81c061ae37",  
  "order_time": "2019-05-05 00:22:01.078"  
}
```

If you remember, user_id is the hash key in the orders table, and order_time is the range key. By querying with the help of both keys, we can fetch any record from the orders table. From here onward, we will send similar JSON to the queue so that we can easily parse the user_id and order_time using JavaScript.

Once you clicked Send Message, you can open CloudWatch logs either by opening the CloudWatch service in the AWS console or by clicking the View logs in CloudWatch button in the Monitoring tab of the lambda function we have created. By navigating from the Lambda function's Monitoring tab, we can open the logs of the order orchestrator lambda function directly. If you open the CloudWatch service to find logs, you have to click Logs in the left-hand navigation switcher. This shows a list of log groups. You should find a log group named /aws/lambda/order_orchestrator. Click it to see the log streams associated to our lambda function. Click the first latest log stream if there are more than one to see the most recent activity. It opens a list of events recorded from the execution of the lambda function, as shown in Figure 2-17.

| | | Time (UTC +00:00) | Message | | | | |
|---|----------|-------------------|---|--|--|--|--|
| 2019-05-19 | | | | | | | |
| ▶ | 16:24:45 | | No older events found at the moment. Retry. | | | | |
| ▶ | 16:24:45 | | 2019-05-19T16:24:45.754Z 9b85e512-7719-4bd0-aa2e-fe543370631 Loading function | | | | |
| ▼ | 16:24:46 | | START RequestId: 9b85e512-7719-4bd0-aa2e-fe543370631 Version: \$LATEST | | | | |
| ▶ | 16:24:46 | | 2019-05-19T16:24:46.504Z 9b85e512-7719-4bd0-aa2e-fe543370631 SQS message 19dd0b57-b21e-4ac1-bd88-01bbb068cb78: {"\user_id": "ce191ec0-6cb-11e9-83d6-d881c0610e37", "\order_time": "2019-05-05 00:22:01.078"}, | | | | |
| ▶ | 16:24:46 | | END RequestId: 9b85e512-7719-4bd0-aa2e-fe543370631 Duration: 808.17 ms Billed Duration: 900 ms Memory Size: 128 MB Max Memory Used: 70 MB | | | | |
| ▶ | 16:24:46 | | REPORT RequestId: 9b85e512-7719-4bd0-aa2e-fe543370631 Duration: 808.17 ms Billed Duration: 900 ms Memory Size: 128 MB Max Memory Used: 70 MB | | | | |
| No newer events found at the moment. Retry. | | | | | | | |

Figure 2-17. CloudWatch Log Events

If you notice, you can see that each console.log is emitted as an event in the log stream. There are few other events to indicate when the execution has begun and when it has ended and how much memory is used and how many milliseconds it ran and so on. Each event is by default shown as a row which we can click to see the full message wrapped to the viewable area. You can change the default view behavior from rows to text by clicking the radio button above the events list. We can also search for an event using a string by entering it in in Filter Events search box.

Occasionally, you may see a few seconds delay between lambda execution and the logs to appear in the CloudWatch stream. You can click Retry to refresh. However, it is also possible that the next invocation of the lambda function created a new log stream, so go back to the log group page of the lambda function and check if a new log stream is created. If so, open that to see the log streams. Logs are split into streams to manage the portions of log easily. This eliminates the need for searching the entire log for our recent information.

Once the lambda function is executed, go back to the SQS and select the orders_queue. If you look at the available messages, you will see there are no messages in the queue. Can you guess where our message has gone? One benefit of having a Lambda function as the trigger is that if there are no exceptions in the execution of the lambda function, the message is considered as successfully processed and is automatically removed from the queue.

However, in our Big Picture, we have seen that the order is moved across several phases by the orchestrator, which means the same lambda function has to process the order message multiple times to determine if the state has changed. So we should prevent the lambda invocation from deleting the message by throwing an error instead of silently exiting the lambda function. This will ensure that the message is retained in the queue to be processed again in 5 minutes as we have given visibility timeout of 5 minutes for the queue.

Test Event to Mimic Lambda

In order to test our lambda function, we don't need to send a message to the queue every time. This also pollutes our queue with test messages, which is not quite necessary. To keep the queue clean, we need to invoke our lambda function with a test event that simulates a message. AWS Lambda already provides a sample event for an SQS that we can modify and use.

Click the dropdown beside the Test button in the lambda function. You should see a Configure test events option in the dropdown. Click it to open a popup wizard. Click the Event template dropdown in the pop-up and choose Amazon SQS which is most likely listed under Recommended because we have an SQS queue in our triggers. A sample event is populated as a JSON. It has a Records key which is mapped to an array of messages. Each message contains a body, attributes, receiptHandle, and a few other keys. The only attribute that interests us for now is the body. The body attribute contains the JSON we used while sending a message.

We don't need other attributes for our testing, so you can remove all attributes except for the body and give the order id in the body as shown in the following code snippet. You may have to change the user_id and order_time for the next sections where we write code that makes use of these parameters and fetches the data from the DynamoDB orders table, but for now, to invoke it with the existing sample code, this works:

```
{
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "{\"user_id\": \"ce191ec0-6ecb-11e9-83a6-
cd81c061ae37\", \"order_time\": \"2019-05-05
00:22:01.078\"}"
    }
  ]
}
```

You can save the event with any name. Give a value in the Event name text box and click Create to save this test event. Once saved, you can select this test event from the dropdown beside the Test button above the Designer tile. Select the test message you just created and click Test as shown in Figure 2-18.



Figure 2-18. Series of Steps to Create Test Input Event

This should invoke the lambda function and produce a log similar to the one we have seen earlier. Open the CloudWatch log stream to confirm there are no issues. If there is any typo in the key names such as Records or body, our lambda function throws an exception, which you can find in the latest log stream as usual.

Orchestration Logic

As we have the skeleton lambda function ready, it is time for us to move on to the crux of our requirement, orchestration. As I have explained earlier, a message will be queued in orders_queue which will be listened by our orchestrator lambda worker; and based on the current status of the order, it populates another message to one of the four queues we have listed earlier. Here are the high-level requirements of orchestration.

ORCHESTRATION RULES

Orchestration relies on the latest status of the order fetched from the DynamoDB table.

1. If the order status is CONFIRMED, move it to PREP_QUEUE.
2. If the order status is PREPARED, move it to COOK_QUEUE.
3. If the order status is COOKED, move it to PACK_QUEUE.
4. If the order status is PACKED, move it to DELIVER_QUEUE.
5. If the order status is DELIVERED, delete it from the orders queue and set status to COMPLETE.
6. If the order status is CANCELLED, delete it from the orders queue.

These rules don't take into consideration complex scenarios that are likely to occur, but these are enough for our dummy online restaurant.

The previous code snippet generated by the blueprint already iterates over each message. Even though we don't have more than one message in our use case, we still have to iterate the same way because we get an array even with a single message. However, we don't have to extract messageId;

all we need is the body of the message. So the following snippet will fetch the order from the message:

```
for (const { body } of event.Records) {
  let order = JSON.parse(body);
}
```

As the body is a string, we need to parse it to convert it to a JavaScript object. Once we have this as an order object, we can read user_id and order_time like properties of the object. We can use these two properties to construct a query request to fetch the complete record from the orders table:

```
let query_input = {
  TableName : "orders",
  KeyConditionExpression: "user_id = :id and order_time = :ts",
  ExpressionAttributeValues: {
    ":id":order.user_id,
    ":ts":order.order_time
  }
};
```

This request can then be used as input for the query method of the document client. A document client can be constructed as shown in the previous chapter: `let docClient = new AWS.DynamoDB.DocumentClient();`. As there can only be one record with the given user_id and order_time, we will always get one item in the response of the query. The AWS object can be created by requiring the aws-sdk module at the beginning of the lambda function. We can make use of promises as shown in the following format to process the order returned by the query promise:

```
let query = docClient.query(query_input).promise();
await query.then(orders => processOrders(orders.Items))
  .catch(error => handleError(error));
```

The logic of handling each order is located in the processOrders method to write modular code. We have also used await keyword so that the lambda function waits until the promise returns. If the query promise successfully returns zero or more items, we call the processOrders method; and if there are any issues executing the query, the handleError method is called. For now, it is as simple as rethrowing the error.

Our code logic of orchestration lies in the processOrders method. If we look at the first rule, if the order status is CONFIRMED, it should be queued in another queue named prep_queue, and the status should be changed to PREP_QUEUE. Let us just focus on writing just one of the orchestration rules. The rest of them will be the same. The following code of processOrders function does the job:

```
processOrders = async (orders) => {
  if (orders && orders[0]) {
    let order = orders[0];
    if (order.order_status === 'CONFIRMED') {
      await pushToQueue(order, process.env.PREP_QUEUE_URL);
      await setOrderStatus(order, 'PREP_QUEUE');
      throw new Error('Order InProgress')
    }
  }
};
```

This is partial code for the processOrders function. We only execute the orchestration logic if there is at least one order in the orders list. If the order status is CONFIRMED, we push it to the queue using a queue URL and set the status of the order in DynamoDB through method setOrderStatus. You may find two new aspects of the source shown in the preceding text. We have used a variable process.env.PREP_QUEUE_URL, which is an environment variable we can pass when a lambda function is being executed. We can configure any environment variables in lambda configuration right below the source code tab, as shown in Figure 2-19.

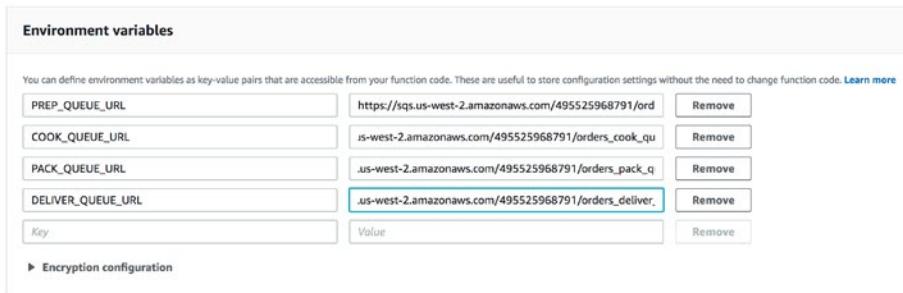


Figure 2-19. Environment Variables to Avoid Hard-Coding in Lambda Functions

Another new aspect is that we are throwing an exception to make sure that the order is not removed from the orders queue. The status of the order is verified again in five minutes, and necessary action is taken. You already know how to write code for functions pushToQueue and setOrderStatus. The following snippet covers the complete orchestration logic. I have also modified the snippet as shown in Listing 2-6 to make use of the switch case to keep the code clean.

Listing 2-6. javascript-sources/chapter-2/demo-02/lambda-function-handlers/orchestrator-lambda-handler.js

```
processOrders = async (orders) => {
  if (orders && orders[0]) {
    let order = orders[0];
    switch(order.order_status.toUpperCase()) {
      case 'CONFIRMED':
        await setOrderStatus(order, 'PREP_QUEUE');
        await pushToQueue(order, process.env.PREP_QUEUE_URL);
        throw new Error('Order In Progress');
    }
  }
}
```

```
case 'PREPARED':
    await setOrderStatus(order, 'COOK_QUEUE');
    await pushToQueue(order, process.env.COOK_
QUEUE_URL);
    throw new Error('Order In Progress');
case 'COOKED':
    await setOrderStatus(order, 'PACK_QUEUE');
    await pushToQueue(order, process.env.PACK_
QUEUE_URL);
    throw new Error('Order In Progress');
case 'PACKED':
    await setOrderStatus(order, 'DELIVER_QUEUE');
    await pushToQueue(order, process.env.DELIVER_
QUEUE_URL);
    throw new Error('Order In Progress');
case 'DELIVERED':
    await setOrderStatus(order, 'COMPLETED');
    break;
case 'PREP_QUEUE':
case 'COOK_QUEUE':
case 'PACK_QUEUE':
case 'DELIVER_QUEUE':
    throw new Error('Order In Progress');
case 'COMPLETED':
    console.log('Removing order as the order is
already completed: %j', order);
    break;
default:
    await setOrderStatus(order, 'CANCELLED');
    console.log('Removing order from orders queue
as status is invalid: %j', order)
}
```

```
    } else {
        console.log('No orders for the given message, it will
be removed from the queue')
    }
};
```

This code expects that you have created the queues prep_queue, cook_queue, pack_queue, and deliver_queue and their URLs are configured on the lambda function as environment variables with names PREP_QUEUE_URL, COOK_QUEUE_URL, and so on. As you may have noticed, we don't have consumers for the other queues, so the order will not move from the PREP_QUEUE stage to the PREPARED state automatically. If you want to test the functionality, make sure you move the status manually.



Note Did you observe the difference between the code snippets? In the first snippet of the processOrder method, we first call setOrderStatus and then push to queue. In the second snippet, we do it in reverse. Can you identify the behavioral difference between them?

EC2 NodeJS SQS Worker

In the previous section, we have seen serverless implementation of a consumer. Serverless has its benefits. However, there can be cases where having a server is a better option. As Lambda invocations are charged per GB-Second, they are going to be cheaper when the throughput is less. However, if the number of requests grow, we will be able to process them on an actual server.

If each request on an average is consuming 0.25 GB-Seconds, we use all the free tier at 0.62 invocations per second 24×7 as the free tier allows you to consume 400,000 GB-Seconds per month. If our average rate of invocations is doubled, we will spend around \$7 per month. So if our

application TPS is 8 for the whole month, the cost for running our Lambda will be around \$35. Of course, for an online restaurant, an average TPS of 8 means millions of active customers every month.

The costs of compute will actually be very small compared to other costs like data transfer. Even though the cost of running everything serverless is less, there can be other use cases that will push the cost of running serverless to be higher. We can run a server that can take the same load at much less price. It doesn't have other costs such as warm-up and so on, and we can make use of server-side caching to get performance optimizations. It is always a cost vs. return comparison.

In this section, we are going to create a NodeJS-based worker running on an EC2 machine. EC2 is a virtual cloud infrastructure service offered by AWS. Creating EC2 instances on AWS gives us the flexibility to build complex applications than possible with lambda or serverless approaches. These instances allow us to build applications just like an on-premise infrastructure, and every aspect is under our control. Creating an SQS worker on an EC2 instance allows us to run the high throughput at a lower cost. It is also possible to run multiple services on a single EC2 machine making full use of its cost.

Launching an EC2 Server

Open the EC2 service from the AWS Management Console and click the ***Launch Instance*** button on the EC2 dashboard page under the Create Instance section. Creating an EC2 instance can be done in a few simple steps. The first step is to select an Amazon Machine Image (AMI), as shown in Figure 2-20, which is a preconfigured template for operation system, application servers, and applications.

CHAPTER 2 SIMPLE QUEUE SERVICE

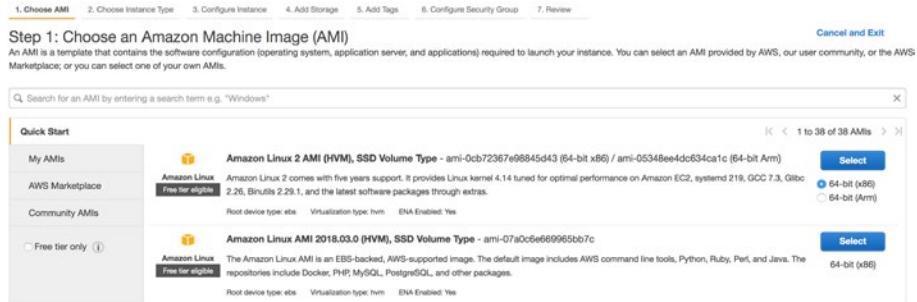


Figure 2-20. Various AMIs Available in the Quick Start Tab

AWS provides many templates to choose from, or even you can start from a base template and create an image with additional configurations required for your setup. Whatever image you select is used as the image for your EC2 instance. So you can pick any one of the images marked as free tier eligible. There are several free tier-eligible images for you to use.

I'm choosing Amazon Linux 2 AMI of the flavor 64-bit (x86) as at the time of writing this book, not many instance types eligible for free tier work with Arm. Click the Select button beside the chosen AMI. I suggest that you stick with this Linux because the commands we use may be slightly different if you go for other OSs like Windows. Even in Linux, each variant may have slightly different commands in a few cases.

Selecting an image takes us to the page where we can select an instance type. An instance is a virtual machine with predefined RAM, CPU, storage, and other parameters. You can relate it to building a desktop computer yourself or customizing memory, processor, and hard drive capacity when buying a computer from an online store. The instance type t2.micro is free tier eligible. It has very low RAM and computing power, but for our use cases, it is more than enough horsepower, and choosing any other instance type means you have to pay for them.

The remaining pages allow you to configure other nonessential details. But as I mentioned earlier, creating an EC2 instance gives us an always running server. To do any useful work on it, we need to be able to log in to

that machine, so we need to configure inbound and outbound rules for network traffic. For us to log in to the server, we can use SSH, but only if we allow it. To do that, click the 6. Configure Security Group tab header in the Launch Instance wizard.

You may not have any security group, so select the first option to create a new security group. A security group is a set of firewall rules. We can open ports of our choice to be accessed from a specific IP address and so on. By default, a rule may have been filled in the table below the description field as shown in Figure 2-21.

| Type | Protocol | Port Range | Source | Description |
|------|----------|------------|------------------|----------------------------|
| SSH | TCP | 22 | Custom 0.0.0.0/0 | e.g. SSH for Admin Desktop |

Warning
Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

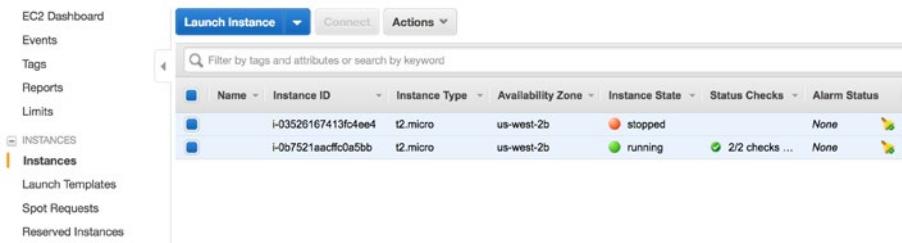
Figure 2-21. Configuration to Allow Inbound Traffic Through SSH

If you don't see any rule for SSH, click the Add Rule button to add one. Choose SSH in the type column dropdown, and the port range is automatically filled with 22. The Source column allows us to restrict the login from a specific IP address or a range. Let us leave it at the default value. The value 0.0.0.0/0 indicates that we can log in to your EC2 instance from any computer. It is good enough for our use cases, and we will be sidetracking a lot if we have to cover all the security best practices. So let us stick to the defaults.

⚠ Caution Allowing the inbound traffic from everywhere is a serious security loophole. Never use it in production and read about AWS networking best practices.

Click the **Review and Launch** button at the bottom of the wizard to finish creating an EC2 instance. You will have to create an SSH key pair to securely log in to the instance you just created. Provide a name **ssh-ec2** and click Download key pair. A file named **ssh-ec2.pem** is downloaded to your computer. Logging in to the EC2 instances can be passwordless if we use the keys. So keep a copy of this file at a secure place where others can't access. Click Launch Instance once you select the checkbox that reminds you to download the key if not done already.

Once done, you will be redirected to a Launch Status page that notifies how far along is AWS in the process of creating our EC2 instance. Click the View Instances button at the bottom of the page to open a list of instances you created as shown in Figure 2-22. Alternatively, you can also go to the EC2 service again and click Instances on left-hand navigation menu. As you can see, I have two instances: one of them is stopped, and the other one is running. In your case, you should have only one if you have never created any EC2 instances earlier.



| Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks | Alarm Status |
|------|---------------------|---------------|-------------------|----------------|----------------|--------------|
| | i-03526167413fc4ee4 | t2.micro | us-west-2b | stopped | None | |
| | i-0b7521aacfc0e5bb | t2.micro | us-west-2b | running | 2/2 checks ... | None |

Figure 2-22. Instances Tables Give Information About All the Instances in Your Account

It is also possible that you may see the instance state as pending for a few minutes as it will take a few minutes for EC2 to spin up a new instance. A few status checks are also confirmed once the instance is up and running. In this table of instances, you will find a few other information necessary for us to log in – not the least, public IP address or DNS name.

We can log in to the host, using this information coupled with the SSH private key file we downloaded earlier before launching the instance. To log in to the host, we can use the following command. The option **-i** is to provide an identity file. The ec2-user is the default user of the instance we created using the quick start AMI:

```
$ ssh -i [identity_file] [user]@[Public DNS Name or IP Address]
```

Example:

```
$ ssh -i ssh-ec2.pem ec2-user@34.219.244.189
```

This logs you in to the EC2 instance we launched earlier. If you see any message asking you to install all updates by running the command `sudo yum update`, do so. This ensures that your instance is secure and stable. You should be able to see the following screen upon login, as shown in Figure 2-23.

```
➜ $ ssh -i ssh-ec2.pem ec2-user@34.219.244.189
Last login: Sun May 26 05:59:08 2019 from c-71-227-158-154.hsd1.wa.comcast.net

              _\   _\_
              _\   (   /
              __\_\_\_|_  Amazon Linux 2 AMI

https://aws.amazon.com/amazon-linux-2/
[ec2-user@ip-172-31-33-45 ~]$
```

Figure 2-23. SSH into an EC2 Instance

Setup Prerequisites on Server

If we want to run a NodeJS application on this server, we need to first install one on the machine or choose an AMI that comes with NodeJS installed. However, installing it is an easy process. Once you install NodeJS and other software necessary for your project, you can create a new AMI for it so that you can help other teams in your organization with the same needs to expedite their server setup.

Install NodeJS

As we wanted to build a NodeJS-based SQS worker, in this example, the first item we need to install is NodeJS. We can install NodeJS by following the instructions from the web site <https://nodejs.org>. But to keep the long story short, you can install by running the following commands in the given sequence:

```
$ curl --silent --location https://rpm.nodesource.com/
setup_10.x | sudo bash -
$ sudo yum install -y nodejs ← To install nodejs from the repo
$ node -v                      ← To verify installation
```

The first command installs a Node.js 10.x repository onto the EC2 instance. Without this, the second command cannot find where to locate the NodeJS package to install from. The final command should give the version number of the NodeJS installed. You can try another version of NodeJS by changing the setup_10.x to a different file corresponding to the version you want to install.

There are other mechanisms to install NodeJS, such as Node Version Manager (NVM). NVM allows us to easily maintain different versions of NodeJS on a single computer and switch between them with a single command. However, we won't need that flexibility, for now, so we are going to skip that.

Install Required NodeJS Packages

To build an SQS consumer, we first need to install the aws-sdk package using npm, which means node package manager. Let us first create an empty NodeJS project by running the `npm init` command inside a folder named `prep_queue_consumer`. Leave all the values at defaults. This creates an empty node project with a `package.json` file in it. In this package, we can install aws-sdk by running the following command in the project folder. This command also adds the dependency in the `package.json` file:

```
$ npm install -s aws-sdk
```

With the aws-sdk dependency, we can write a consumer in the usual way as we have seen in the previous section. However, there is a neat little package named `sqs-consumer` that simplifies processing SQS messages by handling polling at regular intervals and few other simplifications for the process of consumption. You can install that package by running the following command in the same folder:

```
$ npm install -s sqs-consumer
```

While we are installing NodeJS packages, I also want you to install another package named pm2 globally by running the following command. I will explain why we need it shortly. You need to run it as sudo as the ec2-user may not have access to folder `/usr/lib/node_modules`:

```
$ sudo npm install -g pm2
```

Source Code for Consumer

The package `sqs-consumer` is very easy to use. I will show how to do that in this short section. As with many other NodeJS snippets, we need to create an object of the utility by requiring it in our JavaScript file. Once a consumer object is imported, we can create a consumer object by using the `create` method defined in the API of `sqs-consumer`. The following snippet shows an example of how to use long polling to fetch messages in regular intervals:

```
const { Consumer } = require('sqs-consumer');
const app = Consumer.create({
  queueUrl: '...',
  handleMessage: (message) => { // Handle message },
  waitTimeSeconds: 20
});
app.start();
```

The function `handleMessage` is called for every message received from the queue. This is not the extent of configuration possible while creating the consumer. There are few other parameters we can pass while invoking the `create` method. Alternatively, we can also use `handleMessageBatch` instead of `handleMessage` to handle multiple messages at once. Other properties like `visibilityTimeout` and `attributeNames` work just like SQS API. The final line in the code snippet starts the consumption. Until and unless we invoke the `start` method, the consumer doesn't listen to the queue specified in the given URL.

Before we delve further into the source code, let us understand what we want to do in this consumer. Nothing too fancy, all we want to do in this consumer is read the message from `prep_queue` and change the order status to `PREPARED` and log message in the console. We can achieve any complex operations such as invoking a service that starts a machine to collect ingredients and wait for it to finish its work, but the modus

operandi doesn't change. The following implementations achieve the simple functionality we aim to tackle:

```
handleMessage: (message) => {
  let order = JSON.parse(message.Body);
  console.log('Processing the message %j', message.Body);
  setOrderStatus(order, "PREPARED");
}
```

We begin the handle message by parsing the message body into an order object. Once we log the message, we delegate the responsibility of updating the DynamoDB to the setOrderStatus method. The function setOrderStatus constructs an update request with an update expression that sets order_status to whatever input is passed to it.

The snippets shown here must be very much familiar to you based on your previous examples in the book. I have also skipped other parts of snippets that handle authentication using Cognito to get credentials and so on. There are other mechanisms to provide credentials such as EC2 roles to skip authentication, but I'm not going to go over them in the interest of time. Listing 2-7 illustrates the complete code for this demo.

Listing 2-7. javascript-sources/chapter-2/demo-03/sqs-worker.js

```
const { Consumer } = require('sq-s-consumer');
const AWS = require("aws-sdk");
AWS.config.region = "us-west-2";

AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "us-west-2:eac50233-9edd-4082-a12c-
    16120e9b6902"
});

let docClient = new AWS.DynamoDB.DocumentClient();
```

CHAPTER 2 SIMPLE QUEUE SERVICE

```
const app = Consumer.create({
  queueUrl: 'https://sns.us-west-2.amazonaws.com/
    495525968791/orders_prep_queue',
  handleMessage: (message) => {
    let order = JSON.parse(message.Body);
    console.log('Processing the message %j', message.Body);
    setOrderStatus(order, "PREPARED");
  },
  waitTimeSeconds: 20
});

app.on('error', (err) => console.error(err.message));
app.on('processing_error', (err) => console.error(err.message));

setOrderStatus = (order, status) => {
  let request = {
    TableName: "orders",
    Key: {
      'user_id' : order.user_id,
      'order_time' : order.order_time
    },
    UpdateExpression: 'set order_status = :s',
    ExpressionAttributeValues: {
      ':s' : status
    },
    ReturnValues: "UPDATED_NEW"
  };
  let updatePromise = docClient.update(request).promise();
  updatePromise.then(data => console.log(data))
    .catch(err => console.log(err));
};

app.start();
```

Save this code as sqs-worker.js, but before you can run this, change the queue URL to the URL shown in the Details tab of your SQS service, and Cognito identity should be of a user that has SQS access on your account. You can save this with any file name as long as you adjust your command to run the file. We can run this code using the following command. This is a standard mechanism for running any JavaScript file in NodeJS. Run this command in the node project we created and that has the sqs-worker file:

```
$ node sqs-worker.js
```

Running this command starts the consumption, and the process keeps running. As soon as we make an order, the order orchestrator sends a message to prep_queue and the handle message method is invoked and the order status is updated to PREPARED in the DynamoDB table.

If you make an order, in a few seconds, you should see a message similar to the following. The user_id, order_time, and other variable values may change; but the order status before change should be PREP_QUEUE and the after the change should be PREPARED:

```
Processing the message "{\"order_status\":\"PREP_QUEUE\",  
\"food_id\":\"100\",\"user_id\":\"d9-fb54\", \"order_time\":  
\"2019-05-26 18:53:28.732\", \"total_amount\":11.15}" finished  
{ Attributes: { order_status: 'PREPARED' } }
```

In the update record request to DynamoDB in the setOrderStatus method, we requested to return only the values that got changed; hence, we will only get order_status value after update. You can control this by setting the value for parameter ReturnValues in the update request. A few examples of valid values are ALL_NEW and ALL_OLD that will print the entire record.

I should also bring up another aspect of using sqs-consumer. This package will delete the SQS message by default once it has been handled successfully. If you want to retain the message in the queue for any reason, you can throw an exception just like our previous example of the order orchestrator. For our prep_queue example, the message will be deleted from prep_queue, not the orders_queue, so no more work is required from us.

Process Manager to Run the Consumer in the Background

On the face of it, our job looks like it is done. But there is a caveat we need to handle. The consumer keeps running as long as we don't close the terminal window used to SSH into the EC2 instance and start running sqs-worker.js. As soon as we close the shell, the processes created by the shell also get killed. We can run the preceding command with an &! at the end to run the process in the background and keep the process detached from the current shell. However, there is a better mechanism to do the same because it is not reliable.

If you remember, I asked you to install an additional package named pm2 globally along with aws-sdk and sqs-consumer. I mentioned that I would come back to it later. Now is the time for us to use it to keep our prep_queue consumer up and running. The following command starts the consumer in the background using the process manager:

```
$ pm2 start sqs-worker.js --name worker
```

The parameter --name is optional. We can use it to access the process later. If we don't give any name, the process name by default will be the same as the JavaScript file name. Running this command should give an output similar to the screenshot in Figure 2-24. It gives details like process id, status, app name, and so on.

```
[ec2-user@ip-172-31-33-45 prep_queue_consumer]$ pm2 start sqs-worker.js --name worker
[PM2] Spawning PM2 daemon with pm2_home=/home/ec2-user/,pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /home/ec2-user/prep_queue_consumer/sqs-worker.js in fork_mode (1 instance)
[PM2] Done.
```

| App name | id | version | mode | pid | status | restart | uptime | cpu | mem | user | watching |
|----------|----|---------|------|------|--------|---------|--------|-----|---------|----------|----------|
| worker | 0 | 1.0.0 | fork | 7127 | online | 0 | 0s | 0% | 24.4 MB | ec2-user | disabled |

```
use `pm2 show <id/name>` to get more details about an app
[ec2-user@ip-172-31-33-45 prep_queue_consumer]$
```

Figure 2-24. Start the Consumer Using the Process Manager

The advantage of running the consumer using the process manager is that we can make sure that it is always up and running and, moreover, we can also see the logs generated by this process by running the following command. Change the app name if you have used a different one:

```
$ pm2 logs worker
```

This command tails the logs, and you should be able to see the logs of the consumer, which means we should be able to see each time a message is processed. The screen should look like shown in Figure 2-25.

```
[ec2-user@ip-172-31-33-45 ~]$ pm2 logs worker
[TAILING] Tailing last 15 lines for [worker] process (change the value with --lines option)
/home/ec2-user/.pm2/logs/worker-out.log last 15 lines:
/home/ec2-user/.pm2/logs/worker-error.log last 15 lines:
0|worker | Processing the message {"order_status":"PREP_QUEUE","food_id":"300","user_id":"d31974d0-7fe5-11e9-8fb5-c5e519e8c2f4","order_time":"2019-05-26 19:57:40.383","total_amount":9.75}
0|worker | finished
0|worker | { Attributes: { order_status: 'PREPARED' } }
```

Figure 2-25. Tailing Logs Using the PM2 Logs Command

You can safely exit the logs command by pressing Ctrl-C on the screen. This will not disturb the running worker process. The worker will be running even when you exit the SSH and close the terminal used to log in to the EC2 instance. As long as the instance is up and running, the messages will be consumed from prep_queue.

What if you want to kill the worker process because of whatever reason? You can use stop or kill commands of pm2. The kill command will ensure that all processes started with pm2 are killed, whereas the stop command takes the app name as the input:

```
$ pm2 kill
```

Or

```
$ pm2 stop worker
```

Consumption approaches vary slightly based on the choice of platform. If you are building a Java consumer, you can use threads to run consumption logic if you don't want to use any other technologies. Alternatively, you can use Spring or Spring Boot that allows you to create scheduled applications. If you are looking in Python, you may use a package like schedule.

Irrespective of the platform used, fundamental concepts stay the same. We build an SQS client and invoke the receiveMessage API. Once we receive a message, we process it and send a delete message request if everything is in order, or we leave the message so that it will be retried.

FIFO (First-In-First-Out) Queues

In the previous sections of this chapter, we have learned about standard queues. Standard queues allow communication between different applications at scale, but there is no guarantee of ordering of messages. Assume you are creating an application that receives commands from the user that is sensitive to the order in which they are executed. Assume you are building an application that takes buy and sell commands for stocks through a queue. Assume the user wanted to buy the stock first and sell immediately.

When the user sends these calls at quick succession, because of the way standard queues operate, there is no guarantee that the buy message is pulled before sell message. If that happens, our application will try to sell a stock that the user doesn't own, leading to a catastrophe on his financial statement.

Imagine a user is trying to deposit and withdraw some money. If we perform these in the wrong order, we can charge overdraft fee because withdrawal operation is performed before we deposit any money in the account. So as you might guess, the ordering of message is processing supercritical in financial applications. There can be many other applications where the ordering of messages is essential to the domain. To facilitate these use cases where ordering is important, AWS has introduced a new type of queues named FIFO queues or first-in-first-out queues. Another use case is where the duplicates can't be tolerated.

Creating FIFO Queues

For the most part, creation of FIFO queues is no different. The overall process of initiating the queue creation is the same as in the AWS console. Open the SQS console, and click the Create Queue button. This opens up a wizard for creating queues. You are presented with two types of options, and the default option is Standard Queue. Switch to FIFO Queue as shown in Figure 2-26.

As you may notice, the name of the queue is suffixed with `.fifo`. If you don't, the wizard is going to complain about that. The Create Queue button will be disabled until you provide a name that ends with it. If you click the Configure Queue button, you would get exact same options as a standard queue, except for a checkbox to enable content-based deduplication. Clicking the **Create Queue** button completes the creation of a queue just as usual.

CHAPTER 2 SIMPLE QUEUE SERVICE

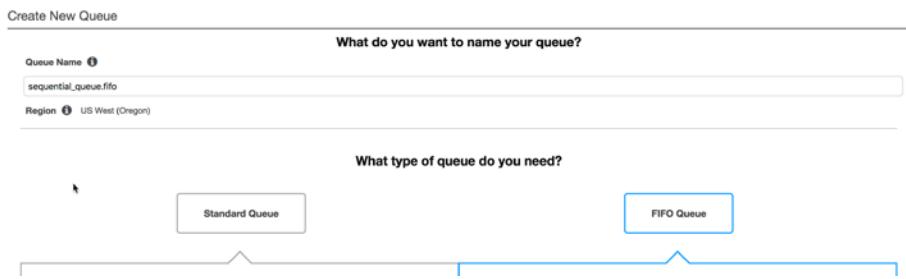


Figure 2-26. Creating a FIFO Queue

If we want to create the same queue using SDK, we can do so by giving the name of the queue that ends with .fifo and passing an additional attribute `FifoQueue` with a value of true. The following code does the job without any functional difference. It returns the QueueUrl if all goes well:

```
var params = {
  QueueName: 'fifo-queue-1 fifo',
  Attributes: {
    'FifoQueue' : 'true'
  }
};

sqS.createQueue(params).promise().then(data => console.log(data))
  .catch(err => console.log(err))
```

Sending Messages

Sending messages to a FIFO queue is the same except for a few differences from a standard queue. You don't need to make any code changes to the API calls. The mandatory differences in request params are as follows:

1. Delay seconds is no longer valid while sending a message – A FIFO queue can only have the delay time at queue configuration. Individual messages can't have this property. If you are trying to migrate from a

standard queue to a first-in-first-out queue, you need to remove this parameter from the input parameters.

2. Message group id – Every message you send to a FIFO queue must contain a message group id. All the messages that have same message group id are ordered. It is okay to give the same message group id for all messages.
3. Deduplication – If you remember, I mentioned an attribute named content-based deduplication while creating the queue. Either this has to be enabled or we need to provide a message deduplication id in the input while sending a message to a FIFO queue.

By keeping all these in mind, we can send messages to a FIFO queue using the following code snippet. There is no difference in the function of QueueUrl, MessageBody, and MessageAttributes. The two additional parameters as you can see are MessageGroupId and MessageDeduplicationId:

```
var params = {
    "QueueUrl": "https://sqs.us-west-2.amazonaws.com/
                  495525968791/fifo-queue-2 fifo",
    "MessageBody": "test message body",
    "MessageGroupIdMessageDeduplicationId
```

```
sqS.sendMessage(params).promise()
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

Because of the message deduplication id, if we send two messages with the same deduplication id in less than a 5-minute timespan, they are considered as the same message. Running the preceding code will produce a response similar to this. This is similar to the response of a send message operation for an ordinary queue except for a sequence number:

```
{
  ResponseMetadata: {
    RequestId: 'c39c6e3b-ce4e-577d-b37f-3b5d08bbe50a'
  },
  MD5OfMessageBody: 'fec7e229adaf3f501ccbeec7b9b291e',
  MD5OfMessageAttributes: '1d4bf50cd0476c6c48b937f1991dc08',
  MessageId: 'c0fac7ff-0317-4faa-8b6c-80cc97867a20',
  SequenceNumber: '18845853310473455872'
}
```

The sequence number is a large number that SQS assigns to each message in a FIFO queue. It is not a consecutive number. If you send the same message twice (same deduplication id) in a span of five minutes, both the requests return the same sequence numbers. If the timespan increases by more than 5 minutes or if each message has a different deduplication id, their sequence numbers are guaranteed to be different, but they most likely are not going to be consecutive numbers.

Receive Messages

In a standard queue, we do not have control in the order in which we receive messages. Irrespective of the order in which the messages are sent, they may be received out of order. In a FIFO queue, we can send messages

to a message group. Whenever we pull messages from a FIFO queue, there is no way to pull messages from a specific message group, but all the messages coming from a message group will be returned in the order they were inserted.

Take the example of messages in Figure 2-27. A producer process keeps producing messages in different message groups. The producer process has a control on which group the message goes to. The consumer process can just request for a message from the queue.

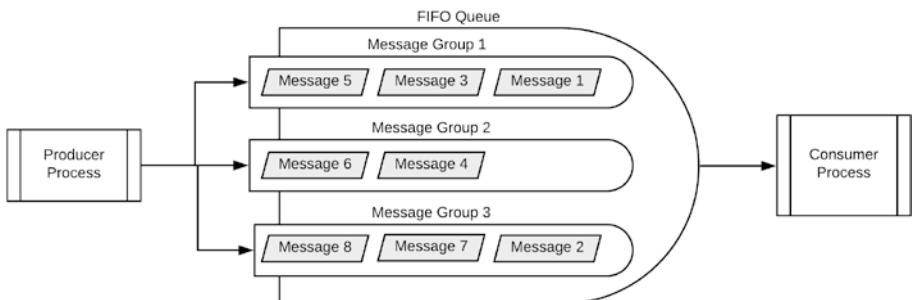


Figure 2-27. FIFO Message Groups

On first request for pulling a message, the consumer may get message 1, but until we process the message and delete it from the queue, message 3 and message 5 are blocked. Irrespective of however many receive message calls are sent to the queue, the consumer may receive messages from message groups 2 and 3. After visibility timeout, message 1 will be returned again if not deleted by then. The code to get messages is pretty much the same:

```
let params = {
  "QueueUrl": "https://sqs.us-west-2.amazonaws.com/495525968791/
    fifo-queue-2 fifo",
  "AttributeNames" : ["Body", "MessageGroupId"],
  "WaitTimeSeconds" : 3
};

let promise1 = sqs.receiveMessage(params).promise();
```

Retry Failed Receive Requests

It is a common occurrence that a receive message request fails due to some network issues or such. If it fails, the messages chosen for returning to the consumer are hidden from subsequent calls. SQS provides a mechanism to receive the same set of messages even if such failure occurs, even if the message visibility timeout is not expired.

The attribute `ReceiveRequestAttemptId` can be passed along with other parameters such as `QueueUrl` and `WaitTimeSeconds` in the receive message request. This parameter is only applicable for FIFO queues. The maximum length of a `ReceiveRequestAttemptId` is 128. It can only contain alphanumeric characters (a-z, A-Z, 0-9) and some special characters (!"#\$%&'()*+,-./;:<>?@[\\]^_`{|}~).

A `ReceiveRequestAttemptId` can be reused for just 5 minutes. Even if we use the same id after 5 minutes, we may not get the same set of messages. You can generate a receive request attempt id using any mechanism such as universally unique identifier (UUID) and use it in the receive message requests. If we don't provide this parameter, SQS auto-generates it.

Dead Letter Queues

As you now know, SQS allows us to retry failed messages with ease. This is an excellent way of building resilient applications at scale. Irrespective of how good the feature is, we can fall into the trap of infinite retries if we are not careful with it.

Imagine a scenario where the upstream application sends corrupt messages to the queue for an hour because of an issue with recent deployment and we have reverted the deployment as soon as we identified it and the upstream application starts pushing good messages to the queue. Consumers can process successful messages as ever, but the

corrupt messages keep failing all the time. Another example can be that a user placed an order from our awesome restaurant application and while the order is in the queue, a developer accidentally deleted some order ids from the DynamoDB table. Every time the order orchestrator or other queue listeners get the message with deleted order ids, the consumers try to fetch the record from DynamoDB, but they keep failing.

As the new messages keep pouring in and the old messages keep failing to increase the load on the consumers unnecessarily, it is a good idea to remove the corrupt messages from the queue. For that, we need to be able to isolate the messages that fail a lot more than a certain threshold. To enable the feature, SQS has a configuration named redrive policy on a queue.

Simply put, the redrive policy is a configuration that specifies a target queue to which a message must be redirected after the message fails to be processed for a fixed number of times. The target queue is called as dead letter queue (DLQ). When the receive count exceeds maxReceiveCount in the source queue, the message is removed from the source queue and is inserted to the dead letter queue.

We can set a redrive policy for a queue anytime, either while creating it or at a later point in time through configuring it. While creating through the AWS console, instead of clicking the Quick Create Queue button in the Create New Queue wizard, you need to click Configure Queue to expand other configuration options. If we want to change redrive policy after the queue has been created, select the queue of your choice, and choose **Queue Actions > Configure Queue**. This opens up a configuration page for the queue, as shown in Figure 2-28.



Figure 2-28. Redrive Policy

You need to select the checkbox titled Use Redrive Policy, to enable a redrive policy. It requires two inputs. The first one is the dead letter queue name, and the second one is the maximum receive count after which the message must be redirected to the dead letter queue. Remember that the dead letter queue should be created before it can be used in a redrive policy. I prefer giving a suffix of '_dlq' for all the dead letter queues of standard queues. You may follow your own conventions that work in your team.

Based on the configuration we have used here, if an order message is received for more than 120 times by either our orchestrator or any other consumer for the queue, the message will be redirected to the queue orders_queue_dlq. If you want to create a new queue with a redrive policy using SDK, you need to pass an additional attribute to request params of the createQueue method as shown in the following code snippet, Listing 2-8.

Listing 2-8. javascript-sources/chapter-2/demo-04/create-queue-with-dlq.js

```
let redrivePolicy = {
  deadLetterTargetArn: 'arn:aws:sqs:us-west-2:495525968791:
  orders_queue_dlq',
  maxReceiveCount: 120
};
```

```
var params = {
  QueueName: 'programmatic-queue-2',
  Attributes: {
    'DelaySeconds': '5',
    'RedrivePolicy' : JSON.stringify(redrivePolicy)
  }
};

sqS.createQueue(params).promise()
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

We need to pass the redrive policy as a string, which can be done with ease by creating a redrive policy object and converting it to a string using the `JSON.stringify()` method when necessary. To modify already created queue, we can use the `setQueueAttributes` operation as shown in Listing 2-9.

Listing 2-9. javascript-sources/chapter-2/demo-04/configure-queue-with-dlq.js

```
let reDrivePolicy = {
  deadLetterTargetArn: 'arn:aws:sqs:us-west-2:495525968791:
orders_queue_dlq',
  maxReceiveCount: 150
};

let params = {
  QueueUrl: url,
  Attributes: {
    'RedrivePolicy' : JSON.stringify(reDrivePolicy)
  }
};
```

CHAPTER 2 SIMPLE QUEUE SERVICE

```
sqS.setQueueAttributes(params).promise()
    .then(data => console.log(data))
    .catch(err => console.log(err));
```

The code, as shown in Listing 2-9, is pretty much the same except for the need to pass the URL of the source queue instead of the queue name as we did while creating a queue. Even the code in other platforms is eerily similar except for a few changes. For example, take a look at this code in Python that sets the attributes using boto3:

```
import json
import boto3

sqS = boto3.client('sqS')

queue_url = 'https://sqS.us-west-2.amazonaws.com/495525968791/
            orders_queue'
dead_letter_queue_arn = 'arn:aws:sqS:us-west-2:495525968791:
orders_queue_dlq'

redrive_policy = {
    'deadLetterTargetArn': dead_letter_queue_arn,
    'maxReceiveCount': '120'
}

sqS.set_queue_attributes(
    QueueUrl=queue_url,
    Attributes={
        'RedrivePolicy': json.dumps(redrive_policy)
    }
)
```

Once the configuration is in place, we need not do anything else. As soon as the message is received for more than the maximum receive count, the message is pushed to the dead letter queue and is removed from the source queue.

Handle Messages in Dead Letter Queues

We have seen how to configure redrive policy to send messages to dead letter queues. However, what's next? What should happen to the messages that are redirected to DLQs? It is entirely up to us. A dead letter queue for all intents and purposes is the same as any other queue. We can create consumers on this queue just like any other queues and handle error scenarios.

For example, if our message goes from orders_queue to orders_queue_dlq, we don't want to maintain the order status as CONFIRMED or any other valid state. Instead we can change the state of the order to ERROR and log so that we can read the log files and debug later.

Summary

In this chapter, we have learned about the first ever service Amazon released and that kick-started the AWS, even though nobody coined the phrase Amazon Web Services. The service Simple Queue Service makes it easy for us to build decoupled components with ease.

If we need to explain the service in as few words as possible, SQS allows us to create queues on Amazon cloud and send messages to them. Any consumer that needs to get the message makes a request to the queue, and one or more messages are returned to the consumer. Once the consumer processes the message, it will be deleted from the queue by a separate request initiated by the consumer.

We began the chapter by understanding how to create a queue. We looked into various approaches to do that. We used CLI, SDK, and the

simplest method of all, using the AWS console. As the creation of a queue is not something that we perform on a daily basis, the AWS console is good enough for this purpose.

Once the queue is created, we have seen how to send messages to the queue with the help of queue URL we have received when we created it. We can send a body and few attributes along with the message. Each message we send to the queue has an expiration time. We can also delay the message after pushing it to the queue.

We can use `receiveMessage` API to pull messages from the queue. We have seen examples in various approaches and how to use parameters like `WaitTimeSeconds` to change the behavior from short polling to long polling. This prevents the number of empty responses received by the consumer saving CPU cycles and bandwidth.

We then understood the life cycle of each SQS message and different timers associated with it. Each message goes through several phases since the time it is pushed to the queue until it is consumed and deleted from the queue. We have created different kinds of SQS workers. We have seen how to process the messages and delete them without too much boilerplate code. We have used Lambda to create a serverless consumer and created a NodeJS worker that runs on a server created in the EC2 service.

We have then learned about first-in-first-out queues that are necessary for sending messages in the order in which they are generated. We have learned how to use deduplication id to prevent duplicates from getting queued to the queues. In FIFO queues, messages can be grouped using a message group id. All the messages in a group are received by the consumer in order.

Finally, we have also covered dead letter queues. The dead letter queues are used for preventing infinite retries of failed messages by setting a limit on how many times a message can be consumed or received by a consumer. If the message receive limit is breached, the message is moved from a source queue to a dead letter queue.

CHAPTER 3

Simple Notification Service

We want to make it even easier for developers to build highly functional and architecturally complex applications on AWS. It turns out that applications of this type can often benefit from a publish/subscribe messaging paradigm. In such a system, publishers and receivers of messages are decoupled and unaware of each other's existence.

—Jeff Barr¹

The Simple Notification Service or simply SNS is a notification service that allows fast and time-sensitive message delivery, in bulk, to multiple subscribers in a single instance. This service operates on a push mechanism which allows the sender to broadcast messages via several endpoints like email, SMS, HTTP, and so on. The sender usually creates a topic to which the recipients can subscribe. Let us first understand what a topic is. A topic is nothing but a channel of communication to which the sender or a publisher can publish their messages. This will automatically send the published message to all the receivers who have subscribed to that specific topic.

¹<https://aws.amazon.com/blogs/aws/introducing-the-amazon-simple-notification-service/>

Let us take an example of a radio broadcast. A radio broadcast is essentially a mode of communication which has two ends. One is the publishing end, which would be the radio presenter or any speaker who's delivering the message on the radio. The other end would be the listeners. Now, the radio presenter always publishes his content on a specific radio channel.

The listeners of his content will be the ones who have tuned into that channel. In this scenario, we can say that the radio presenter is the publisher, the channel is the topic, and the listeners are the receivers. We can see here how only the listeners who have tuned into the channel can listen to the presenter's content. This is exactly how SNS works. All the receivers who have subscribed to a topic will receive the message the moment it is published in the said topic.

Difference from SQS

Now that we have a basic idea on how SNS works, let us see how it differs from the Simple Queue Service (SQS). As we discussed in the previous chapter, SQS is a pull-based service where the messages are sent to a queue, waiting to be picked up. This may not be suitable if you want the message to be picked up by more than one recipient or you want the message to be delivered to the endpoint right away rather than waiting for it to be picked up.

Let's take a look at this scenario of a group chat. One of the friends sends a message to the group chat saying they are going camping that weekend. The message simultaneously gets delivered to three other friends who are in the group chat. After receiving the message, one of them starts packing for the trip, one starts making the food, and the other starts assembling all the emergency supplies they might need. We can see

here that immediate and simultaneous message delivery, in this case, has allowed the friends to save time and prepare for the trip more efficiently. In a way, this enabled a scenario of parallel processing. This is only one of the many advantages of SNS.

There could be several such situations in your application where the message has to be delivered to several components at once to initiate parallel processing. We can also consider the scenario where, if something breaks down or is not going as planned, a warning has to be delivered to the team of developers so they can fix it as soon as possible. These are the times SNS can be really handy and efficient when compared to SQS.

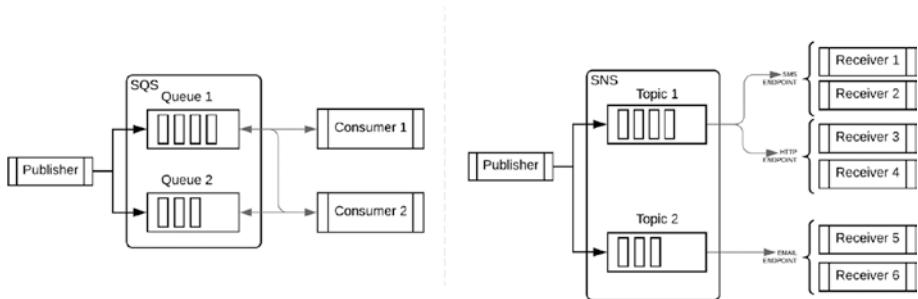


Figure 3-1. SQS vs. SNS

In SQS, each consumer will have to poll for messages, but in SNS, the message will be sent to all the subscribers automatically. SNS supports various types of endpoints like SMS, email, HTTP, and API. This is depicted in Figure 3-1.

Setting Up

Before the SNS topic can be used, we need to create it and subscribe to it. Only then the messages will be delivered to the subscribed endpoints. Creating the topic and subscriptions is a fairly simple process. Even the required code is very minimal. Let us tackle that first.

Creating Topics

As discussed, the publisher, who sends the messages, usually does that via publishing those messages in a topic. Let us see how to create a topic in AWS. Open the Simple Notification Service from the console. If you haven't created a topic before, you can see in the Dashboard that your Topics and Subscriptions are 0. To create a new one, click the Topics hyperlink on the left navigation bar in the console, below Dashboard. Click Create topic, as shown in Figure 3-2.

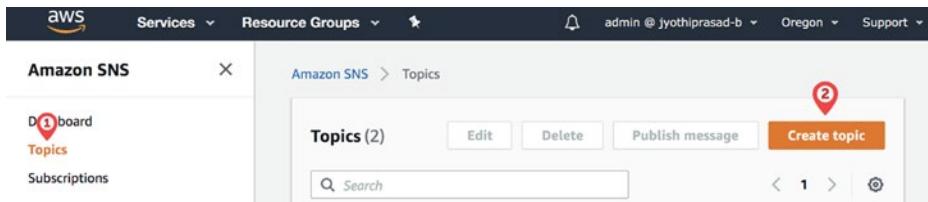


Figure 3-2. Initiate Topic Creation

This will open a page where it asks for all the basic details like the name of your topic, its display name, and several other advanced settings that concern security and access policies of your topic as shown in Figure 3-3.

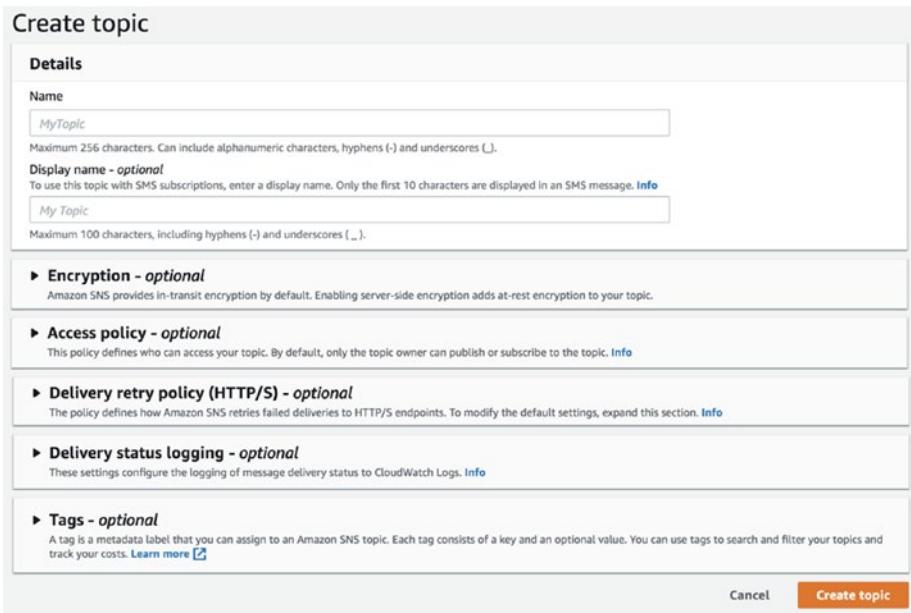


Figure 3-3. Create Topic Wizard

Choose a name for your topic and also a display name. The display name is what will be primarily seen by all your receivers. I am creating a topic with the name OrderAlerts and display name AWSOME-RESTAURANT, as I will be publishing alert messages to this topic. You can always give a name that makes more sense as to what you will be publishing to it, but it is not mandatory. It is solely for convenience purposes.

After giving the name to your topic, you can choose to enable encryption on your topic. Note that Amazon, by default, provides in-transit encryption; and by clicking Enable Encryption, you can choose to avail at-rest encryption which means your message is encrypted as soon as it is received by the service and is only decrypted right before delivery. As we are only looking at creating a simple topic, let's not bother about it now.

The Access policy tab, right below the Encryption tab, helps you control who can publish or subscribe to your topic. You can choose the settings of your choice.

Click the Create topic button at the bottom of the page, and there you have it, your first topic in SNS. You can now see it listed under your Topics if you click the Topics bar. If you want to use CLI to create a topic, the following command works for you. The property name is mandatory, and all attributes are optional:

```
create-topic --name <value> [--attributes <value>]
```

There are different attributes we can pass to the command. Most of them correspond to the properties displayed in UI while creating the topic. Here is an example command. The latest version of CLI may have more CLI parameters such as tags:

```
create-topic --name cli-topic --attributes  
DisplayName=CLI-TOPIC
```

Doing it programmatically is quite simple as well. We just create an SNS client and call the createTopic method on the client. The following is the interface accepted by the method:

```
export interface CreateTopicInput {  
    Name: topicName;  
    Attributes?: TopicAttributesMap;      ← KeyValue pair of  
                                            attributes  
    Tags?: TagList;                      ← Array of KeyValueMap  
                                            of tags  
}
```

The topic attributes map can accept various parameters such as DisplayName, KMS Master Key ID, Policy (access policies), and Delivery Policy. The tags are any key-value pairs that can be used to tag any resources in AWS. Both attributes and tags are optional parameters. Here is the least amount of code you need to write to create a topic:

```
const AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.SharedIniFileCredentials
    ({profile: 'admin'});

let sns = new AWS.SNS();

sns.createTopic({
    Name: "API-TOPIC"
}) .promise()
.then(data => console.log(data));
```

If your user configuration is good, it should create a new topic or update an existing topic and return the topic ARN in JSON format as shown in the following:

```
{
    ResponseMetadata: {
        RequestId: 'e04d54a4-d13b-59c6-a56f-0130fd0f569e'
    },
    TopicArn: 'arn:aws:sns:us-west-2:495525968791:API-TOPIC'
}
```

As soon as you execute this code, you should be able to find the topic in the topics page of the SNS service in the AWS Management Console as shown in Figure 3-4.

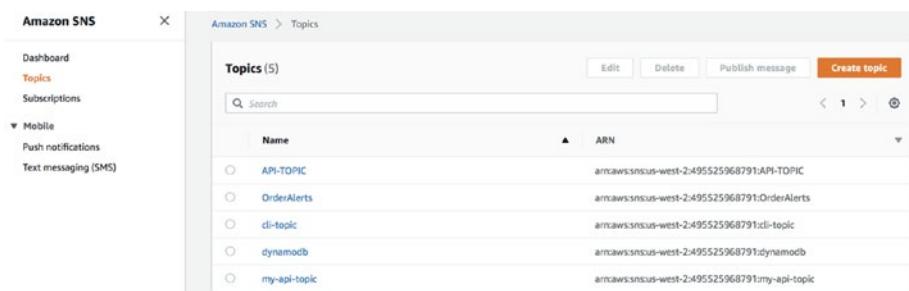


Figure 3-4. Topics List

Create Subscription

Publishing messages to this topic will automatically deliver them all to its subscriptions, as discussed. Before we see how to publish a message to your topic, let us understand how we can create subscriptions. Click the Create subscription button that is found when you click open your created topic. This allows you to create a subscription specific to this topic. The topic's ARN will be auto-filled, so just choose your endpoint from the Protocol dropdown.

Let's start by choosing email for now. Select email, and you will be prompted to enter an email address that you want to add as a subscription. After entering the desired email address, click Create subscription. Now you can see it listed in the subscriptions list in your topic dashboard. Right after you create an email subscription, an email will be sent to the entered email address asking to confirm subscription. Unless the receiver confirms the subscription, you can only see it listed in your subscriptions list as "Pending Confirmation" and cannot send any messages until the confirmation is received.

You can add any other type of subscriptions such as SMS, Lambda, HTTP endpoint, SQS, platform application endpoint, and so on; you can see the status of all your subscriptions in the list of subscriptions if you open any topic as shown in Figure 3-5.

| ID | Endpoint | Status | Protocol |
|---------------------------------------|---|----------------------|----------|
| Pending confirmation | test-email@gmail.com | Pending confirmation | EMAIL |
| 3903c61d-cbf0-4440-bcf4-f6bfb5f2374ca | arn:aws:lambda:us-west-2:495525968791:function:lambdaFunction | Confirmed | LAMBDA |
| ebc3cc40-7789-4f9e-8582-cdb0e839dfd0 | +19999999999 | Confirmed | SMS |

Figure 3-5. Subscriptions List

After the receiver confirms the subscription, an ID is generated for the subscription. Now, whenever you publish a message to your topic, it will automatically be sent to the subscription you have just created, given the case the receiver confirms it by clicking the Confirm Subscription link they receive in email. The following snippet of code shows how to subscribe using API:

```
let params = {
    TopicArn: 'arn:aws:sns:us-west-2:495525968791:OrderAlerts',
    Protocol: 'SMS',
    Endpoint: '+12062222222'
};

sns.subscribe(params).promise()
    .then(data => console.log(data));
```

The Protocol parameter is to indicate what type of endpoint we need to use. It is case insensitive, so you can use email or EMAIL to specify an email endpoint. The email address can be passed through the parameter Endpoint, just like the phone number. If the email doesn't conform to usual email address conventions, the subscribe method throws an exception with message "invalid email address." If you create an HTTP

server to receive the SNS messages, you can pass the IP address and port. For Lambda endpoint, you need to pass the ARN of the Lambda function.

Similarly, you can choose various endpoints like SMS, AWS Lambda, Amazon SQS, and such. By selecting AWS Lambda to be your endpoint, you can choose what lambda function you want to be executed, every time a message is published to the topic.

How do we use this feature to subscribe a user in our awsome restaurant application? We can add a few fields in the About page of our application to accept a phone number and email address and a Subscribe button. When the user enters his phone number and email address and clicks on subscribe, we make an API call to SNS to register the user to the topic. Listing 3-1 shows how the `About.vue` is modified to accept the phone number and email.

Listing 3-1. `javascript-sources/chapter-2/demo-02/awsome-restaurant/src/views/About.vue`

```
<template>
  <div class="about">
    <input v-model="phone" placeholder="phone number"><br>
    <input v-model="email" placeholder="email address"><br>
    <button type="button" @click="on_subscribe_
      click()">Subscribe</button>
    <p>{{subscribe_message}}</p>
  </div>
</template>

<script>
  import store from "../store";
  import { SnsCaller } from "../utils/sns_caller";
  import { ON_SUBSCRIBE } from "../constants";
```

```
let sns_caller = new SnsCaller();

export default {
  name: "about",
  methods: {
    on_subscribe_click() {
      sns_caller.subscribe(this.phone, this.email);
      store.commit(ON_SUBSCRIBE, { phone_number: this.phone,
        email: this.email });
      this.subscribe_message = 'Check your email to confirm
        subscription';
    }
  },
  data: function() {
    return {
      phone: store.state.phone_number,
      email: store.state.email,
      subscribe_message: ""
    };
  }
};
</script>
```

The phone number and email address text fields are bound to parameters, which were sent to the subscribe method of SNSCaller that calls SNS to save the subscription. This subscription logic is triggered when the Subscribe button is clicked by invoking the `on_subscribe_click` method:

```
async subscribe(phone_number, email) {
  await sns.subscribe({ TopicArn: ORDER_TOPIC, Protocol:
    'sms', Endpoint: phone_number })
    .promise().then(data => console.log(data));
```

```
await sns.subscribe({ TopicArn: ORDER_TOPIC, Protocol: 'email', Endpoint: email })  
    .promise().then(data => console.log(data));  
}
```

Along with the subscription logic, we are also using store.js to save the information in the browser's local storage. If you want to look at the whole code, you can refer to the awsome-restaurant source code in chapter-03/demo-02. Once you save the code and run the project with the 'npm run serve' command, go to the About page, enter the phone number and email address, and click the Subscribe button. You should see a message "Check your email to confirm subscription" as shown in Figure 3-6. As always, email subscription is only confirmed after you click the confirmation link you receive in email. Hence, we are displaying the message to check email.



Figure 3-6. Subscription

Publishing Messages

Once we have successfully subscribed, we can start using the topic to send messages to the user. This process is called publishing a message. Let me briefly explain how to publish a message as it's a fairly simple and uncomplicated process. Open the topic you want to send a message to. You can see a Publish message button above the Details tab. Just click the Publish message button to open the wizard as shown in Figure 3-7.

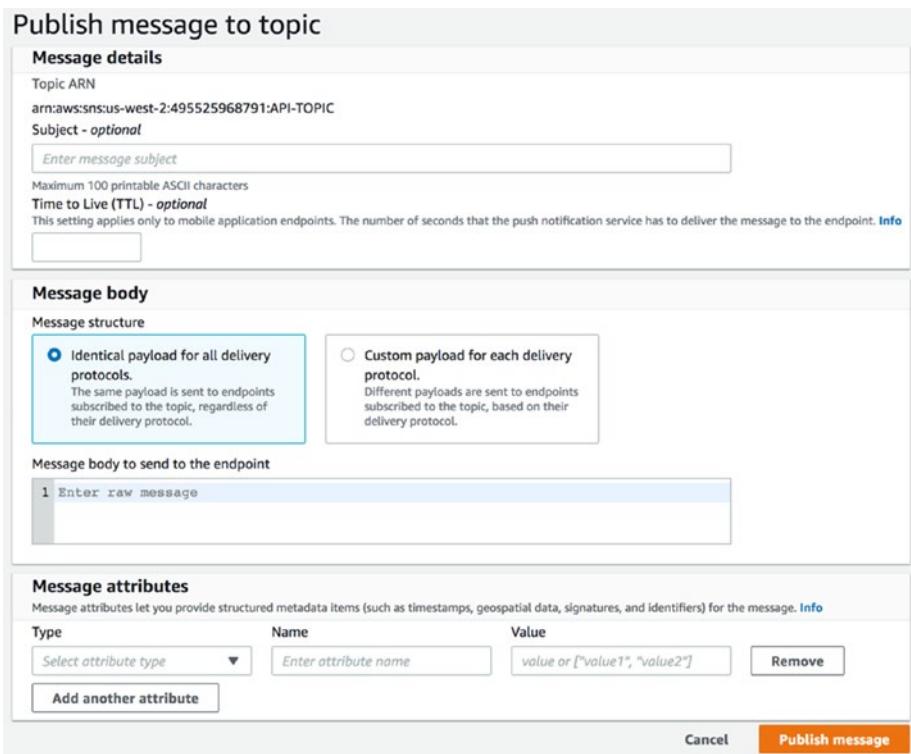


Figure 3-7. Publish Message Wizard

The wizard may appear quite busy, but it is easy enough to understand. The Subject is used in the subject field for an email endpoint. For an SMS endpoint, it is just prefixed to the message body. The TTL property is only applicable for Mobile Application endpoints that consume push notifications. If you are developing an application that sends sports updates of a match whenever something interesting happens, and if the user mobile goes offline for the duration of a game and comes online, he will see a billion notifications. It doesn't make sense to send every notification of the game once when the game is over. Maybe the updates in the last 15 minutes may make sense depending on the situation. Hence, the TTL for all the notifications is 15 minutes or 900 seconds.

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

The tile below Message details is for providing a message body. You first have to select one of the two radio buttons before you type in the message to specify a message structure. The first radio button is selected by default and is for messages that are uniform across the endpoints. Each endpoint gets the same message. The actual payload has to be typed in the text box below the radio button group.

The second radio button changes the message body to be a JSON map. Each key in the map is for one endpoint. You can provide a different message for each endpoint as in the following:

```
{  
    "default": "Order Confirmed",  
    "email": "Your order placed at 2019-07-01 15:53:34 has been  
              confirmed, with items XYZ",  
    "lambda": "user123_2019-07-01 15:53:34",  
    "sms": "Order has been confirmed at 2019-07-01 15:53:34",  
}
```

As you can see, we can send an SMS and email and invoke a lambda function all at once by using just one message, which was not possible with SQS. We will see how to generate these messages programmatically and send them later, but for now, fill in any text you want and move on to the next tile.

The final section of the publish message wizard can be used to provide message attributes, which are metadata associated with each message. These attributes can be used by the message receivers to make a call whether to process that message or not. You can have up to ten attributes for each message. Fill all the details and click the button Publish message located at the bottom of the wizard.

API Method

As easy as sending the message through the AWS console, it is not really useful for building real-world applications. Most of the times, we end up publishing messages programmatically. To do that, we can use the publish method of the SNS client. At the very simplistic level, the code looks like this:

```
publishParams = {
    TopicArn: 'arn:aws:sns:us-west-2:4955134315791:API-TOPIC',
    Message: 'Hello world'
};

sns.publish(publishParams).promise()
    .then(data => console.log(data));
```

We need to pass the topic ARN and message to the publish method. Make sure that you use the ARN of the topic you have created. So the region, account id, and topic name will be different for your code. Upon executing the code after changing the values accordingly and if everything is alright, you get a response like this:

```
{
  ResponseMetadata: {
    RequestId: 'd4a4c8b6-ec19-5373-aa53-9d158f068201'
  },
  MessageId: 'e3f44e47-6f98-5b98-a4b6-1be40c87cae9'
}
```

There are a few other parameters this method accepts. It takes an object of type PublishInput interface which is defined in JavaScript SDK as in the following. As you can see, Message is the only mandatory parameter. You have to provide one of TopicArn, TargetArn, and PhoneNumber. TargetArn parameter is used for platform endpoints.

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

```
interface PublishInput {  
    TopicArn?: topicARN;  
    TargetArn?: String;  
    PhoneNumber?: String;  
    Message: message;  
    Subject?: subject;  
    MessageStructure?: messageStructure;  
    MessageAttributes?: MessageAttributeMap;  
}
```

The remaining fields can be correlated with the UI fields. If you want to send a different message for each endpoint, the value of the `MessageStructure` should be “JSON.” The message payload has to be passed in the `message` attribute. You may have noticed that there is no TTL field in the interface. Here is a sample message with other parameters such as `MessageAttributes` and `MessageStructure`:

```
publishParams = {  
    TopicArn: 'arn:aws:sns:us-west-2:495525968791:API-TOPIC',  
    MessageStructure: 'json',  
    Message: JSON.stringify({  
        default: 'Do not worry, order is being prepped',  
        email: 'Order is confirmed'  
    }),  
    MessageAttributes: {  
        "user_id" : {  
            DataType: "String",  
            StringValue: '1234567890'  
        },  
        "food_ids": {  
            DataType: "String.Array",  
            StringValue: '["100","200"]'  
        }  
    }  
}
```

```

        }
    }
};

sns.publish(publishParams).promise()
    .then(data => console.log(data));

```

In this snippet, we passed two different types of attributes. One is a string, and the other is string array. The message is also a JSON object that is intended to send one message for email and a different one for all other types of endpoints. To test this code, create the topic with an email subscription and an SMS subscription. When you execute this code, you should receive an email with text 'Order is confirmed' and an SMS with text that says 'Do not worry, order is being prepped'. The message attributes are not visible in either case.

With this knowledge, we can modify our FoodItem.vue to invoke a method that publishes a message to OrderTopic. This topic can act as a trigger for the Lambda function that both saves the data in DynamoDB and sends an SMS to the user simultaneously. The simultaneous communication is taken care of by SNS; we just have to publish the message to the topic. Create a new topic named OrderTopic and save its ARN in constants.js in awsome-restaurant project. We can create a helper class that publishes the message to OrderTopic, as shown in Listing 3-2.

Listing 3-2. javascript-sources/chapter-3/demo-02/awsume-restaurant/src/utils/sns_caller.js

```

const AWS = require("aws-sdk");
import { AWS_REGION, COGNITO_IDENTITY_POOL_ID, ORDER_TOPIC } from "../constants";
import store from "../store";

```

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

```
export class SnsCaller {
    constructor() {
        AWS.config.region = AWS_REGION;
        AWS.config.credentials = new AWS.CognitoIdentityCredentials({
            IdentityPoolId: COGNITO_IDENTITY_POOL_ID
        });
    }

    async save_order(food_id, amount) {
        let payload = {
            user_id: store.state.user,
            food_id: food_id,
            order_time: new Date().toISOString().
                replace("T", " ").replace("Z", ""),
            total_amount: amount,
            order_status: 'CONFIRMED'
        };

        let params = SnsCaller.getParams(ORDER_TOPIC, payload);

        await SnsCaller.getSnsClient().publish(params).
            promise();
        return payload;
    }

    static getParams(topic, payload) {
        return {
            TopicArn: topic,
            MessageStructure: 'json',
            Message: JSON.stringify({
                lambda: JSON.stringify(payload),

```

```

        sqs: JSON.stringify(payload),
        default: 'Order is confirmed'
    })
};

}

}

```

There are a few changes in our approach. With SNS, we don't have a way to get what is the `order_time` back in response. Previously, as the Lambda function is generating the `order_time`, it can send that info back once the execution is over. In the SNS world, there is no processing of the message. Even when the endpoint is Lambda, there is no way for us to get the response back from lambda. Hence, we create the entire payload, including order time. With this, the Lambda function has to store the received payload in the database without bothering to do any more magic.

In the payload, we are passing the entire order object for SQS and lambda endpoints, but for the remaining endpoints, we pass a simple message. As we are passing multiple endpoints' JSON, we need to pass the message structure parameter as JSON.

This class is used just like the `lambda_helper` when the Order button is clicked in UI. The Order button is part of the `FoodItem` component; hence, the starting point of the execution is in the `FoodItem.vue` file. Previously, we were invoking `LambdaCaller's save_order` method; now we would be calling SNS caller's `save_order` method. So we can bring in SNS interaction with minimal code changes. For the major part, there are barely any changes. Just the `lambda_caller` is changed to `sns_caller`:

```

let sns_caller = new SnsCaller();
sns_caller
  .save_order(this.item.id, this.item.price)
  .then(t => {
    console.log("success");
    store.commit(ADD_ITEM, [this.item.id, t]);
  })

```

```
    this.item.ordered = !this.item.ordered;
    this.item.errorred = false;
  })
  .catch(err => {
    console.log(err);
    this.item.errorred = true;
});
});
```

You can refer to the complete source code in demo-02 of chapter-03 in javascript-sources. You need to add a lambda function to the subscription for this order payload to be processed and added to the database. Let us see how to create the receiver.

Message Receivers

We have seen how to set up a topic and send messages so far. However, that is just half the game. For endpoints like SMS and email, there is nothing much to be done. Information gets to the user, and he reads it and does whatever he wishes to. But that is not the case all the time. We do have Lambda endpoints, SQS endpoints, HTTP endpoints, and application endpoints. Let us first see how to create a receiver that can process the message. We shall create a lambda function whose trigger is SNS.

Lambda Receiver

You are already familiar with creating a lambda function with name notification-receiver. Create one and add an SNS trigger. It asks you to provide the ARN of the SNS topic. Once the ARN is added, it should appear as shown in Figure 3-8. You also have to make sure that the lambda function has sufficient access to interact with DynamoDB and so on. All this lambda function has to do is save the data to DynamoDB.

So scroll down in the Lambda page until you find a tile that shows the IAM role associated with that lambda function and click the link that says view the role in the IAM console. Search for the policy that states DynamoDB Full Access and attach it to the role. You should be pretty familiar with this now, so I'm not going into the details of this.

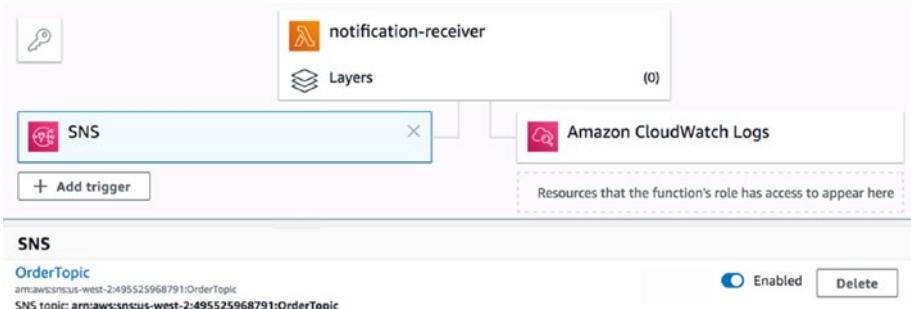


Figure 3-8. Lambda Receiver Before Adding DynamoDB Access

Now for the meat of the receiver, we need to implement the lambda function handler to process the incoming event. Whenever the SNS sends a notification to the Lambda function, it takes the following form if only one message is sent to the lambda function. If multiple messages are received, the records array contains multiple items:

```
{
  "Records": [
    {
      "EventSource": "aws:sns",
      "EventSubscriptionArn": "arn:aws:sns:us-west-2:495525968791:OrderTopic:a52600b",
      "EventVersion": "1.0",
      "Sns": {
        "Message": "{\"user_id\":\"760f7533-a5bc-11e9-b76c-d5045d3c917f\", \"food_id\":200, \"order_time\":\"2019-07-14 23:50:37.143\", \"total_amount\":12.45}",
        "TopicArn": "arn:aws:sns:us-west-2:495525968791:OrderTopic"
      }
    }
  ]
}
```

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

```
    "MessageAttributes": {},  
    "MessageId": "53f7fdef-7bb8-5fac-b8ff-a84ab984bebc",  
    "Subject": null,  
    "Timestamp": "2019-07-14T23:51:24.357Z",  
    "TopicArn": "arn:aws:sns:us-west-  
                2:495525968791:OrderTopic",  
    "Type": "Notification",  
    "UnsubscribeUrl": "https://sns.us-west-  
                      2.amazonaws.com/?Action=Unsubscribe&  
                      SubscriptionArn=arn:aws:sns:us-west-  
                      2:495525968791:OrderTopic:a52600b"  
  }  
}  
]  
}
```

As you can see, the message we send is property Message. As we didn't pass any message attributes when the Order button was clicked, the MessageAttributes parameter is empty. The Lambda function can process each message. I have removed a few parameters like signature to save some space. Our lambda function can parse the message attribute to get the order information to save in DynamoDB. Here is the complete source code for the lambda handler in Listing 3-3.

Listing 3-3. javascript-sources/chapter-3/demo-02/notification-receiver-lambda-handler.js

```
let AWS = require('aws-sdk');  
let ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
exports.handler = async (event) => {
    for(let record of event.Records) {
        let sns = record.Sns;
        let order = JSON.parse(sns.Message);
        let item = {
            user_id: { S: order.user_id },
            order_time: { S: order.order_time },
            total_amount: { N: "" + order.total_amount },
            food_id: { S: "" + order.food_id },
            order_status: { S: "Confirmed" }
        };
        let params = {
            TableName: "orders",
            Item: item
        };
        await ddb.putItem(params).promise();
    }
    const response = {
        statusCode: 200
    };
    return response;
};
```

In this code, we iterate on the event.Records array. We are parsing the JSON object from the message attribute and creating an item and calling the putItem method on the DynamoDB client. Now, you can run the application and click the Order button, and the Lambda function should be called along with receiving an SMS.

SQS Receiver

Our previous architecture of order orchestration expects that a message is pushed to queue “orders_queue.” With our current change of simplifying lambda just to add data to the orders table, how can we make sure that our architecture doesn’t fall apart? As you may have noticed, we can also have an SQS endpoint. We can subscribe an SQS queue to any SNS topic with ease, as shown in Figure 3-9.

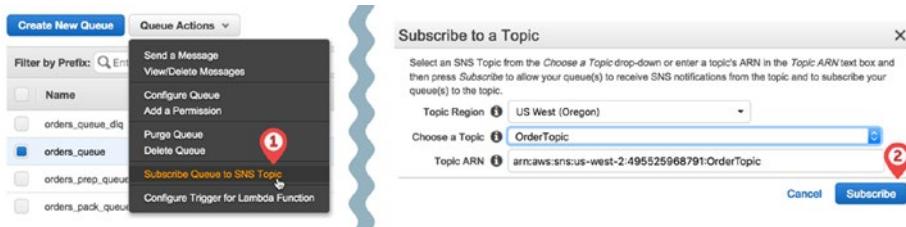


Figure 3-9. Configuring a Queue to Receive Messages from a Topic

We can select the queue in the SQS console and add a subscription by choosing Queue Actions ➤ Subscribe Queue to SNS Topic. This brings up a wizard where you can choose from all the topics you have in the region. Once you have selected the topic, you can click the Subscribe button to finish the formality. You can subscribe more than one SQS queue to the same topic at a time. Select all the queues you want before following the same steps.

Once the subscription is successful, you can then publish a message to the SNS topic and can view the message in the SQS queue immediately. Once the message lands in the queue, all the usual rules apply. The message gets retained for the configured time of retention period and gets retried as per the settings.

As our previous code snippet is sending the entire order object in a string form for the SQS endpoint, we would be getting it in Message property of the SQS message pushed to the queue. The following is the subset of attributes of an SQS message that is posted to an SQS queue from an SNS topic. I have removed a few other fields such as signature, signature version, and unsubscribe URL:

```
{  
    "Type": "Notification",  
    "MessageId": "aff1854b-29cd-5923-b7de-d0e5d66d7187",  
    "TopicArn": "arn:aws:sns:us-west-2:495525968791:OrderTopic",  
    "Message": "{\"user_id\": \"7ad8fb23-ac12-11e9-82f7-  
                b97930f6c453\", \"food_id\": 200, \"order_  
                time\": \"2019-07-22  
                00:03:35.744\", \"total_amount\": 12.45}",  
    "Timestamp": "2019-07-22T00:03:44.079Z",  
}
```

Any client polling on the preceding queue is supposed to handle reading from the Message property to get the actual content, and for supporting any more features like unsubscribing, it has to parse to unsubscribe URL property and make an HTTP call so that the SQS queue will not get any more messages from SNS. However, this is a rare scenario and most likely not used. One scenario you may use it is if there is a per-user queue and we allow the user to opt out of a certain process.

Message Filtering

A few sections ago, we have seen how to subscribe to a topic for receiving all kinds of messages through numerous endpoints. However, there is a minor flaw. We have one OrderTopic for all users of our restaurant web site. If our application ever goes to production, there will be more users,

and if all of them subscribe to the same topic, each user gets an SMS for all the orders in our application.

This is definitely not what you expect from a notification service. SNS does provide a mechanism to filter messages based on a policy. This policy can be set at the subscription level. We can set this policy when we create it, or an existing subscription can be modified to add a policy. Let us first see how to add a subscription filter policy.

A subscription policy allows us to specify attribute names and a list of values associated with the attribute. SNS evaluates each message against attributes linked to it before deciding on which subscriptions should receive the message. A message may have more attributes that are not present in the subscription filter policy; such attributes are ignored from the decision-making process. A subscription accepts a message under the following conditions:

1. Any message attributes that are not in both the message and policy are ignored.
2. Any attribute that is of BinaryType is ignored.
3. If an attribute is present in both the policy and message, the values of the attribute are matched.

To add a subscription policy, open the topic in the SNS console, and click the subscription link. In the subscription page, click the Edit button as shown in Figure 3-10. Clicking the Edit button opens the edit wizard where you can enter a subscription policy in JSON format.

In the figure, I have entered a subscription policy that has one attribute `user_id` that has an array which contains one value. I have entered the UUID that my browser has created when I opened the application for the first time. You should enter the `user_id` created by your application. Now, every time a message is sent with the `user_id` attribute, it will be accepted by the subscription if the value of the attribute is the same as the value mentioned in the policy.

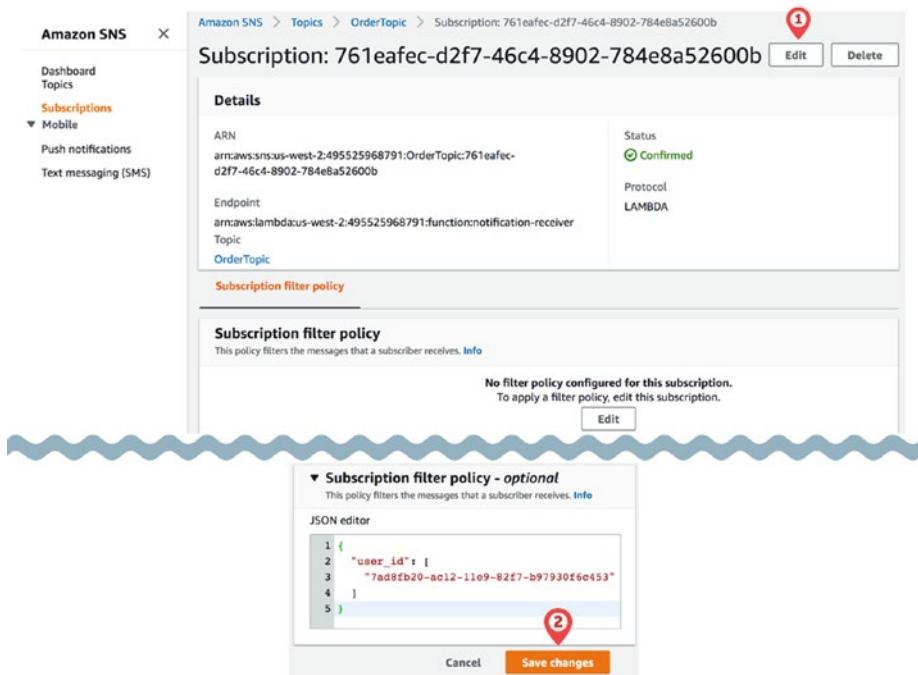


Figure 3-10. Editing Subscription to Add Filter Policy

Here is the code snippet that shows how to add the filter policy while creating a subscription through API. The subscribe method accepts a phone number or email. The filter policy object is the same irrespective of the endpoint type. It just has attribute user_id, which has the same value as the user id we stored in the browser storage:

```

subscribe(phone_number, email) {
  let filterPolicy = {
    user_id: [store.state.user]
  };

  if (phone_number) {
    await SnsCaller.getSnsClient()
      .subscribe({

```

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

```
    TopicArn: ORDER_TOPIC,
    Protocol: "sms",
    Endpoint: phone_number,
    Attributes: {
        FilterPolicy: JSON.stringify(filterPolicy)
    }
})
.promise()
.then(data => console.log(data));
}

if (email) {
    await SnsCaller.getSnsClient()
    .subscribe({
        TopicArn: ORDER_TOPIC,
        Protocol: "email",
        Endpoint: email,
        Attributes: {
            FilterPolicy: JSON.stringify(filterPolicy)
        }
    })
    .promise()
    .then(data => console.log(data));
}
}
```

In this code, we create the filter policy JSON object and pass it through the FilterPolicy attribute in the attributes map of subscribe input parameters. This policy has to be supplied for each subscription we create on the topic for the user. We are also making sure that phone number and email fields are not empty before adding subscription.

At the other end of the spectrum, we need to make sure that we add the attribute that matches this filter policy. We have already seen how to attach attributes to a message while publishing it. The following snippet adds the message attribute to send the message as per our previous filter policy. The name of the attribute must be `user_id`, and the data type is a string, and the value should be UUID:

```
getPublishParamsWithFilter(topic, payload) {
    return {
        TopicArn: topic,
        MessageStructure: "json",
        Message: JSON.stringify({
            lambda: JSON.stringify(payload),
            sqs: JSON.stringify(payload),
            default: "Order is confirmed"
        }),
        MessageAttributes: {
            user_id: {
                DataType: "String",
                StringValue: store.state.user
            }
        }
    };
}
```

With this change, our application can support sending notifications to multiple users without spamming inboxes of our customers as the message each user receives is relevant to their order alone, which was not the case earlier. The subscription policy expects that we have an attribute with name `user_id`, so our new code satisfies that criterion.

Filter Expressions

The previous example is just one way to represent a filter. Sometimes, we may need a much sophisticated mechanism for the decision-making process. Subscription filter policy JSON allows specifying much complex filtering criteria.

Whitelisting

You have already seen this approach. The previous syntax of specifying a string array is for mentioning a one-of condition. When we provide multiple string values in the filter policy for an attribute, the message must have the same attribute but should contain one of the values. The following is the example filter policy for specifying a whitelisted value:

```
{ attribute_name : [ "value 1", "value 2", "value 3" ] }
```

This policy can match any of the following message attributes. At least one of them must be passed along with the message.

```
{ attribute_name : { DataType: "String", StringValue: "value 1" } }
```

Or

```
{ attribute_name : { DataType: "String", StringValue: "value 2" } }
```

For whitelisting numeric values, we need to use the numeric keyword in the policy and an equality operator. The numbers need not be an exact decimal representation. However, for numeric expressions, we can't give more than one whitelisted value using the equality operator:

```
{ total_amount : [ { "numeric" : ["=", 10.5] } ] }
```

Matches

```
{ total_amount : { DataType: "Number", Value: 10.5 } }
```

Or

```
{ total_amount : { DataType: "Number", Value: 1.05e1 } }
```

Blacklisting

There are cases where you may want to specify the values that are not allowed instead of the ones that are allowed. It is especially important when there are many accepted values but very few that are not. In such cases, we can use the anything-but expression to specify the negation. Here are a few examples of anything-but expressions:

```
{ food_id : [ { "anything-but" : [100, 200] } ] }
```

```
{ user_id : [ { "anything-but" : ["abcd", "defg"] } ] }
```

Both these expressions are essentially doing similar things. The first one is for numeric values, and the second one is for strings. When we provide multiple values in the anything-but expression, the message attributes should not contain any of these values. The subscription will receive the message only in that case.

Prefix Matching

When you have to filter based on the prefix of a string value, you may use this prefix matching. The following is an example of prefix expression. Imagine that we set up a subscribe feature to all the new foods we launch in our restaurant. The customer is only interested in receiving a message about new dishes of type Asian. We can set up a filter expression like this.

```
"food_interests": [{"prefix": "asian"}]
```

To satisfy this filter policy, we can prefix all our food items based on the type of dish, such as asian-tempura or indian-biryani. We can add these

attributes to the message when we are publishing about the new dishes to a topic that receives messages related to interest subscriptions. This can match both of the following:

```
"food_interests": {"Type": "String", "Value": "asian-sesame-chicken"}  
"food_interests": {"Type": "String", "Value": "asian-tempura"}
```

Number Range Matching

Imagine that you are running a promotion wherein if a customer orders for more than \$15, you want to give them a special discount of 10%. In such case, you may create a separate queue that listens to this orders topic but only wants to get notified about orders with total amount of more than \$15. For such subscription, we can use numeric expressions as shown in the following:

```
"total_amount": [{"numeric": [">=", 15]}]
```

This expression can match all the messages that have the total_amount attribute with value of more than or equal to 15. Remember that this is an additional attribute we have to pass. The total_amount attribute of the order object is not considered automatically. The following snippet shows how to pass the total_amount attribute:

```
getParamsWithFilter(topic, payload) {  
    return {  
        TopicArn: topic,  
        MessageStructure: "json",  
        Message: JSON.stringify({  
            lambda: JSON.stringify(payload),  
            sqs: JSON.stringify(payload),  
            default: "Order is confirmed"  
        }),  
    },
```

```

MessageAttributes: {
    user_id: {
        DataType: "String",
        StringValue: store.state.user
    },
    total_amount: {
        DataType: "Number",
        StringValue: "" + payload.total_amount
    }
};

```

In this snippet, we are passing the total amount associated with the order when we publish it as a message to an SNS topic. As the filter is a numeric policy, we need to pass the data type as number, even though we pass the actual value as string. So the customer receives the message only when the cost of the order is more than 15\$. It is also possible to specify that the amount should be between two numbers, as shown in the following:

```
"total_amount": [{"numeric": [">", 5, "<=", 15]}]
```

In this expression, we have specified the lower limit as well as the upper limit as opposed to just the lower limit in our previous example. One example where you might find this useful is if you are running a cashback offer when the order amount is between 5 and 15 dollars.

Existence Check

Our SNS topic may receive various kinds of messages; one of them is food orders and food interests of the customer; other things may be new deals. If a user wants to receive only the messages about orders he made, we need to identify if a message is of deal type or of food order or food interest type. For these kinds of checks, we can add another attribute whose value is not important.

Once that attribute is added to the message, our filter policy can check if that attribute exists. Along with the verification of attribute value, we can also check if an attribute exists in the message attributes. To do that, we use the exists expression, as shown here:

```
"total_amount": [{"exists": true}]
```

This filter expression matches a message only if the message contains total_amount, which means every order is passed through the filter. Any message that doesn't contain this attribute is filtered out.

Multiple Conditions

In any reasonably complex applications, we will have more than one condition. A customer may only want to receive messages related to his order and when the order limit is more than \$15, that is actually a combination of two filter conditions. We can mention any number of filtering conditions in a single policy. However, all of them are an evaluation with an AND logic. It means that the message flows through only if the message attributes match all the policies. There is no special syntax to combine multiple filtering conditions:

```
{
  user_id: [store.state.user],
  total_amount: [{ numeric: [">=", 15] }]
};
```

With this filter policy, both conditions should be satisfied by the message attributes. The user id should be the same, and the total amount must be more than or equal to 15. Otherwise, the message is skipped by this subscription.

Push Notifications

In previous examples, we have seen SMS, SQS, email, and Lambda. However, we left out push notifications and HTTP endpoints. In this section, we will see how to get push notifications using Firebase Cloud Messaging (FCM). We will add a functionality to enable push notifications in our awsome restaurant app. Enabling push notifications has a few more steps than other types of endpoints.

SNS cannot send push notifications directly; it needs to make fallback on other push notification services such as Google Cloud Messaging, Firebase Cloud Messaging, Amazon Device Messaging, Apple Push Notification Service, and so on. These push notifications directly appear in the mobile app or web app as alerts, badges, or even sound alerts. These push notification services maintain the connection with the mobile to send on-demand notifications.

The process to send push notifications depends on the notification service. The Firebase Cloud Messaging (FCM) service is from Google. The Google Cloud Messaging service is deprecated in favor of FCM and is no longer available since May 2019. Let us now look at the steps involved at a high level to enable receiving notifications through FCM:

1. Create a Firebase project and app to generate an API key and other config information.
2. In our client application, initialize Firebase messaging using config info from step 1.
3. Generate a Firebase token from our client application.
4. Create a platform application in SNS with the API key from step 1.

5. Create an onMessage handler in the client to run every time a message is published.
6. Create a platform endpoint subscription with the token.

Irrespective of the technology of the client application, the generic steps involved are the same. An Android app may initialize Firebase messaging different from a web application or an iOS app. The actual code to generate Firebase token or handle every message is also going to be dependent on the platform in which you receive the notifications. In our example, I will show how to do that in our web application. Feel free to follow the same steps for the platform of your choice.

Firebase Steps

You need to create a project and app in Firebase as specified in step 1. This can be done by opening the Firebase console by opening the following URL. If you have never created any project in Firebase, you will be given an option to create one on the first screen. Otherwise, make sure you figure out how to create one by exploring:

<https://console.firebaseio.google.com>

This opens up a wizard, as shown in Figure 3-11. You can give a name to your project, and it auto-generates a project id and identifies your location. If you want to change any of them, click the pencil icon beside the fields to edit them.

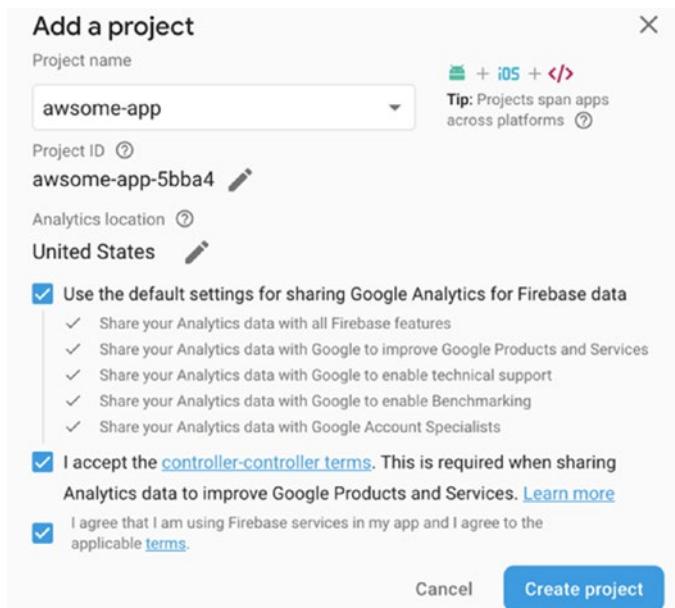


Figure 3-11. Create a Firebase Project

You will have to accept the terms and conditions and settle for the default settings for sharing Google analytics for Firebase data. Once all the prerequisites are met, click the Create project button to finish the proceedings of creating a project. Once done, you will be presented with a wizard where you can create apps.

Each project can be associated with more than one app. If our awsome restaurant is successful, we may create an Android app to allow users to order food with ease, but we still have to support our web users. With Firebase, we can do it with ease. For now, click the web app icon that looks like an empty HTML tag.

A new pop-up is displayed where you can enter a name for the app. Let us enter the name web-app and click Register app. Once done, you can find some HTML and JavaScript code in the same wizard. Make a note of it; you would need it soon. The most important piece of code is as follows. I have ignored the HTML parts of the code. If you observe the following

code, you can see that it has apiKey. We need it to connect the SNS with Firebase:

```
// Your web app's Firebase configuration
var firebaseConfig = {
  apiKey: "AIzaSyAkYXGixsSHzC9Y0TR0w7zNZcBmpfIYpn8",
  authDomain: "awsome-app-1.firebaseio.com",
  databaseURL: "https://awsome-app-1.firebaseio.com",
  projectId: "awsome-app-1",
  storageBucket: "",
  messagingSenderId: "964583283916",
  appId: "1:964583283916:web:b1a18efb6915bb59"
};

// Initialize Firebase
firebase.initializeApp(firebaseConfig);
```

This code initializes a Firebase object by passing apiKey and appId. These are mandatory for our application to prove that it is authorized to receive/send information from Firebase. Once you have this code noted somewhere, you can continue to the console. We don't have much else to do in Firebase unless you are feeling adventurous and want to try some other Firebase features.

Initializing Firebase in web-app

We can use the configuration code we copied from Firebase in the previous section as is. However, we need to make sure that Firebase is available as a dependency in our awsome restaurant. To do that, make sure that you install Firebase dependency as needed using the npm install command. Once all the dependencies are met, add the following lines of code to main.js:

```
import firebase from "firebase/app";
import * as m from "firebase/messaging";

// Config code copied from firebase

messaging = firebase.messaging();

Notification.requestPermission().then(permission => {
  if (permission === "granted") {
    messaging.getToken().then(currentToken => {
      if (currentToken) {
        console.log(currentToken);
        window.token = currentToken;
      } else {
        console.log("No Instance ID token is available.");
      }
    }).catch(err => {
      console.log("An error occurred while retrieving
      token. ", err);
    });
  } else {
    console.log("Unable to get permission to notify.");
  }
});

messaging.onMessage(payload => {
  alert("Message received: " + payload.data.default);
});
```

The first two lines of code are just importing Firebase and Firebase messaging-related components. After imports, paste the code we copied from Firebase in step 1. The configuration constants such as apiKey and message sender id change from project to project and app to app, so make sure that you don't copy from this book. Instead, copy from your Firebase console when you created the project and app.

The pasted config code initializes a Firebase object from which we get a messaging object. We need the messaging object to communicate with the Firebase messaging service. Before we can receive push notifications, we need to get permission from the browser for doing so. This is achieved by the `Notification.requestPermission()` line. Once granted, we can call Firebase Cloud Messaging to get a token.

This token is necessary for Firebase to recognize which browser or app needs the information. The line `messaging.getToken` takes care of getting a device token. If we get a device token, we can store it globally so that we can use this information in `sns_caller` to create a subscription at SNS.

Typically, we may also store this token at the server so that our server-side code can send notifications to a specific device. However, this is out of scope of this book. We have also not written any code for the refreshing token as it does expire after a certain period of time.

The final three lines of code are a foreground handler for the message received event. Let us not do anything fancy here. Let us just show an alert. Whenever we receive a message, we receive a payload that looks like this which is identical for all endpoints. Hence, the JSON path `payload.data.default` fetches the message text from the received payload:

```
{  
  "data": {  
    "default": "Message Text"  
  },  
  "from": "719631067901",  
  "collapse_key": "do_not_collapse"  
}
```

This is a foreground handler. It just gets activated when our application is in the foreground for background worker; we need to code it in a file named firebase-messaging-sw.js. For now, create an empty file with this name in the public folder of our project. It is possible to create this file with another name, but we have to register it. As this is not a book about Firebase, we're not going into the nitty-gritty of Firebase.

SNS Setup

The previous section sums up all our interactions with Firebase. However, that is just half the story. As SNS is our focus topic, we need to wrap things around by configuring this end of the business. Open the console of SNS and click Push notifications under the Mobile section in the left-hand navigation bar. In the Mobile push notifications page, you can create a platform application by clicking the Create platform application button.

In the next screen that looks like in Figure 3-12, you need to give a name to your application. If you have multiple push notification platforms, you need to create separate applications for each of them. Give a name that is concise yet informative such as awsome-firebase-app. This tells us the name of our end product and the notification platform. You may eventually create many more applications that serve multiple push notification platforms. Hence, the name should be clear.

Choose Firebase Cloud Messaging from the dropdown to use Firebase. For Firebase, we need to provide an API key as credentials. Each notification platform needs a different set of credentials. For example, Baidu for Android needs an API key and Client secret. A Windows phone needs a Certificate and private key.

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

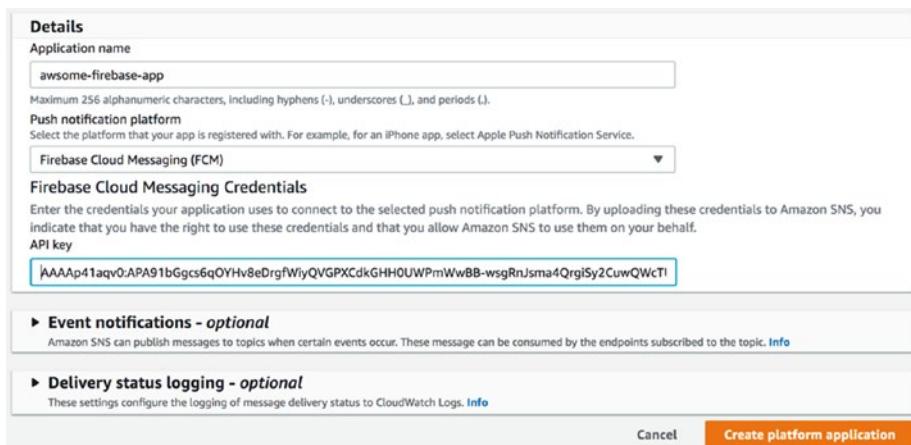


Figure 3-12. Create Platform Application Wizard

There is a catch here. The API key is not the key we copied from the Firebase config object. We actually have to provide the server key that can be obtained from the Firebase console. Go to project settings, which can be exposed by clicking the gear icon in your Project Overview. Go to the Cloud Messaging tab, and you can see the server key. The process is depicted in Figure 3-13. Copy this server key and paste it in the API key field in the SNS Create platform application wizard.

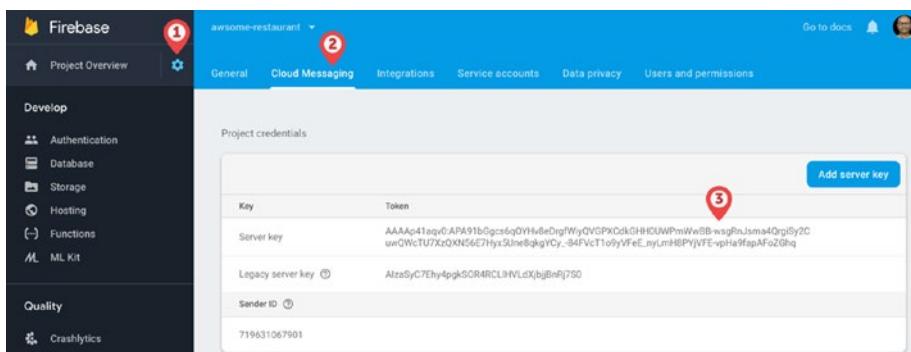


Figure 3-13. Steps to Get the Server Key

The other configurations in this page are optional. The Event notifications section is to keep track of events that occur on this application. For example, whenever a device registers itself to the application, we can configure SNS to send a message to another topic. They may be necessary for us to keep track of metrics of new users and so on.

The final optional configuration section, Delivery status logging, is to configure logging of messages. By default, each message is configured to be also delivered to logs, but sometimes, our application may be sending too many messages which may lead to huge CloudWatch costs. So it may be beneficial to reduce the percentage. If we give the value as 10%, one message of every ten messages is picked for logging. We can also configure IAM roles in the same section.

To finish the process, we need to click the Create platform application button. If you have given the wrong key, you will see an error that indicates invalid credentials. Once the creation is successful, you will end up on a page where you can gather ARN for this app.

An application is analogous to a topic. However, there is one big difference. We can send a message to a topic, and all the subscriptions that have matching filter policy get the message. However, for an application, there are no mass notifications. Each device that needs to be notified has to subscribe using a device token, and a message is sent for each subscription separately. Let us now see how to create a subscription for our web application page opened in your browser.

Client Endpoint

Add the following code in sns_caller.js. This makes a createPlatformEndpoint API call and passes the token we generated and attached to the window through main.js. Make sure that you use the ARN you gathered after creating a platform application:

```
async enablePushNotifications() {
  SnsCaller.getSnsClient()
    .createPlatformEndpoint({
      PlatformApplicationArn: PLATFORM_APPLICATION_ARN,
      Token: window.token
    })
    .promise()
    .then(data => console.log(data));
}
```

How does this method get triggered? For that, we will be adding another button in the About view and invoke this method upon clicking that button. Here is the code that is placed in About.vue. The HTML portion is placed in the template. The JavaScript portion of this code is placed inside the methods attribute of the script tag. For the complete code, open the About.vue file:

```
<!-- Template -->
<button type="button" @click="on_enable_push_notifications_
click()">
  Enable Push Notifications
</button>

// JS Portion
on_enable_push_notifications_click() {
  sns_caller.enablePushNotifications();
  this.subscribe_message = "Push Notifications Enabled";
}
```

Testing It Out

Run the application using the `npm run serve` command. Navigate to the About page; you should see the Enable push notifications button. Clicking this button will show the message “Push Notifications Enabled” if everything has gone right. If your browser shows an error in developer console-like, the browser is not supported. Make sure you use latest versions of Chrome, Firefox, or any other supported browsers. The following screenshot, as shown in Figure 3-14, is a split view of how our application UI and SNS endpoints list should look like after clicking the button.

The figure consists of two vertically aligned screenshots. The top screenshot shows the application's About page. At the top, there are links for 'Home' and 'My Order', and a user ID '12066120252' with an email 'yothiprasadd@gmail.co'. Below this is a 'Subscribe' button and an 'Enable Push Notifications' button. A blue wavy line separates this from the second screenshot. The second screenshot shows the AWS SNS 'Endpoints' list for the application 'awsome-firebase-app'. The 'Endpoints' tab is selected. It displays one endpoint entry:

| Token | User ID | Status | Channel ID | ARN |
|--|---------|---------|------------|--|
| e3HW_LrUgEAP OKTMBtj1BQjtz GJtQcNpNxb7i4 83Fm06o3Gkoty VMF9IDWMEP7ji | - | Enabled | - | arn:aws:sns:us-west-2:495525968791:endpoint/GCM/awsome-firebase-app/41643ea3-1532-33d9-9af0-b3df3ae8c086 |

At the bottom of the screenshot, there are buttons for 'Edit' and 'Delete'.

Figure 3-14. Enabling Push Notifications Creating an Application Endpoint

Once the endpoint is created, it is ready to send notifications to our applications. Select the token to which you want to send the message to enable the Publish message button in the previous screenshot. Click the Publish message button and type in any message. Enter a message body before sending the message. For now, just leave the message structure as

identical payload for all delivery. You are already aware of how to send custom payload for each delivery.

Publishing the message should trigger an alert in our application immediately. This is what we have coded in the `onMessage` method in `main.js`. However, we don't have to restrict ourselves to just showing alerts or notifications. We can trigger any other piece of code. This is how we can use SNS to execute code based on events triggered in the cloud.

Even a HTTP/S endpoint will also go through similar steps to receive notifications or trigger code. However, we won't be covering it in this book in the interest of time. We have to implement the client in a certain way, and this can be learned by going through the AWS documentation.

Miscellaneous Topics

There are few other configurations that we will cover in this section to bring completeness to the knowledge surrounding the Simple Notification Service. Each of the configurations can be accessed by going through separate tabs in a topic, as shown in Figure 3-15. However, they can only be edited by going to the edit mode of any specific topic.

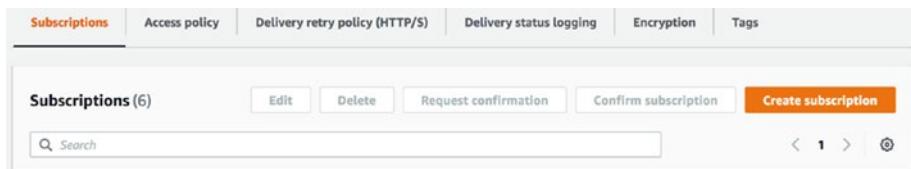


Figure 3-15. Other Configurations

The Delivery retry policy tab is to configure retries in case of a failed delivery for a HTTP/S endpoint. By default, a message is retried three times for this endpoint, but it can be changed along with delays between retries and backoff functions. The other configuration is about logging. We can also enable logging of specific endpoint messages.

Encryption is used to make sure that we don't transmit messages in plain text. Encryption is achieved by using keys created using the KMS service. The same keys are used by other services accessing the message to decrypt the message. Tags are used to monitor costs and usages for each topic.

Access Policy

Usually, the SNS topic can only be accessed by its owner. Only the owner can subscribe or publish a message. By changing access policies, we can control who can and cannot interact with this topic. Open the topic, and go to the Access policy tab. By default, it has all the possible actions listed, and the condition is that the source owner is the same account number as the account in which the topic is created. Here is the modified version of a policy:

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

```
"SNS>ListSubscriptionsByTopic",
"SNS>Publish",
"SNS>Receive"
],
"Resource": "arn:aws:sns:us-west-
2:495525968791:OrderTopic",
"Condition": {
    "StringEquals": {
        "AWS:SourceOwner": "495525968791"
    }
},
{
    "Sid": "__default_statement_ID_1",
    "Effect": "Allow",
    "Principal": {
        "AWS": "*"
    },
    "Action": [
        "SNS>Subscribe",
        "SNS>Publish",
        "SNS>Receive"
    ],
    "Resource": "arn:aws:sns:us-west-
2:495525968791:OrderTopic",
    "Condition": {
        "StringEquals": {
            "AWS:SourceOwner": "892325968791"
        }
    }
}
```

```
    }  
]  
}
```

This policy has two statements, each with a different SID. The effect of both the statements is to allow access to subscribe, publish, and receive actions for any resource of AWS account 892325968791(let us just call it account X). This gives access to the order topic in the account's us-west-2 region of 495525968791. You can have any number of statements, but they should have different statement ids.

With this policy, a lambda function in account X can publish a message to order topic or receive messages. But it is not allowed to delete a subscription or get topic attributes. It is necessary if we want to add a queue from one AWS account to listen to a topic in another account.

Summary

We have gone through various topics related to the Simple Notification Service. Unlike SQS, SNS is used to send messages to more than one receiver at a time. SNS supports various types of endpoints, including push notifications. We have also learned about the differences between SQS and SNS in this chapter.

We have begun the chapter by learning about how to create a topic and subscriptions using the console and API. We have ignored the CLI approach as it is not used frequently. We have also seen how to publish messages to the topic manually through the console and using API. We have also updated our awsome restaurant app to publish a message to our topic when a customer orders food.

So we changed the architecture of our application to rely on SNS topics instead of the queue so that the customer gets an SMS message and email and the lambda function gets called to save order info into the table. By doing so, we have seen how to create lambda receivers and SQS receivers.

CHAPTER 3 SIMPLE NOTIFICATION SERVICE

We have then seen how to control who gets notified when a message is published to a topic. We have utilized subscription filter policies to ensure that only the customer who orders the food gets the message about the order status. We have understood the syntax of subscription filter policies and learned different approaches to control the message delivery through whitelisting, blacklisting, and so on.

Finally, we have seen how to use push notifications. For the push notifications to work, we need to integrate SNS with Firebase or other push notification services. In this chapter, we have used Firebase Cloud Messaging. We have integrated Firebase Cloud Messaging in our web application to receive push notifications from SNS.

In the next chapter, we will see how to use a Simple Workflow Service or SWF to build better workflows instead of relying on disjointed queues or topics. Keeping the orchestration elements such as queues and topics separate, it is difficult to keep track of what is going on in any one order flow. There is no single place for us to see the progress of a specific order irrespective of whether we use SQS or SNS. This problem is resolved when we move to services such as SWF.

CHAPTER 4

Simple Workflow Service

The previous chapters showcased how easy it is to let applications talk to each other. The services allow you to focus on the core logic of the application without worrying about failures or retries. However, the architecture is getting complex. You now have five queues each having their own consumers that process the messages that arrive in the queues. There is an orchestrator that moves an order in a flow based on the order status.

Our application demands the complexity, as each component/consumer performs a task on its own. The message consumers should know when they should start working on the tasks. So we have an orchestrator that is telling the component to begin working by inserting the message in the appropriate queue. Essentially, we are creating a workflow by creating queues. However, if the complexity of the application grows, so does the number of queues. Even though it doesn't cost you in terms of infrastructure, it just costs your developers' sanity. Task orchestration can quickly become chaotic if not kept in check.

If all you need is to build a workflow, AWS has another service for that exact purpose. The Simple Workflow Service (SWF) is a generic solution for building distributed program workflows. It is a cloud-based workflow engine. It takes out the complexity of managing queues and other fault

tolerance mechanisms to ensure the data flows through various stages until it serves its purpose. It provides various features to managing tricky aspects of scheduling, concurrency, and dependencies of all the stages in the workflow. It does the heavy lifting of message passing, locking, and state management.

Architecture of SWF Applications

The applications built using SWF mainly consist of deciders and workers. Deciders are nothing but orchestrators; we have an orchestrator even in Chapter 2 where we discussed SQS. The orchestrators decide which activity is supposed to be executed next. They regularly poll SWF for all the events that happened so far, and based on the history of events, the deciders have the responsibility of identifying the next task and communicating the decision back to SWF.

The workers will then come into play. They get information about the activities to be performed, and they complete the work and communicate the result back to SWF. The cycle then repeats. Another decider will receive the task success or failure event along with the history of events that happened on the same workflow execution and decide which activity has to be executed next. If the decider thinks that re-executing the previous task is necessary, it just communicates the decision to SWF.

Apart from workers and deciders, there is another component that kick-starts the workflow execution to set everything in motion. This is akin to sending a message to `orders_queue`. As I mentioned earlier, we don't have to rely on queues to let our orders data flow through the entire workflow of preparing the order, cooking the order, packaging it, and then delivering it. SWF acts as the message broker in this case. An overall application diagram looks as shown in Figure 4-1.

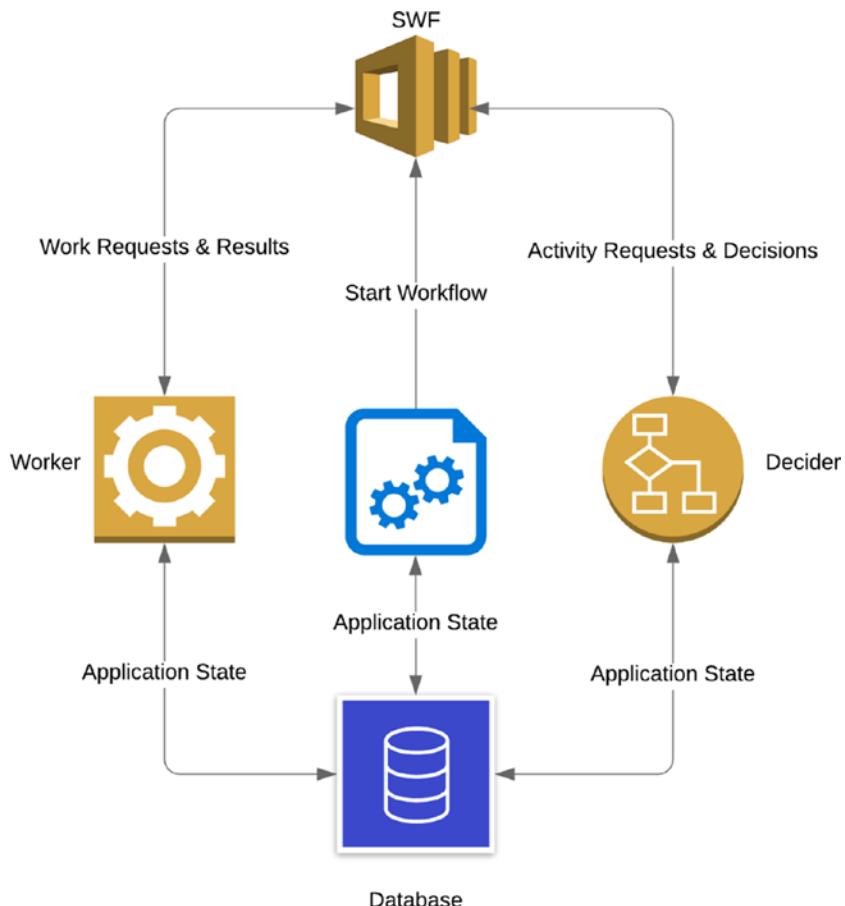


Figure 4-1. Application Architecture Using SWF

The three components of the architecture that execute the code we write are starter, decider, and worker. They are actors and can take any form. They can be NodeJS programs running on an EC2 instance or can be lambda functions or something that is running on-premise, completely outside of the AWS infrastructure. Irrespective of where they run, their jobs don't change. The starter will have to execute code to trigger new workflow execution, and the decider has to be a constantly polling process that gets the history of events from SWF to decide the next event. The workers have to complete tasks scheduled by deciders.

They can share and access the application state to perform business-critical functions or for deciders to decide which activity needs to be performed based on the current state of the object. This is no different from our SQS architecture, except here we don't have to maintain any queues. Because the entire orchestration logic lies in the decider, we can change the logic without having to recreate new queues or such with minimal effort.

Let us look at how we can develop each of the components. We shall first begin with creating a workflow in SWF and see how to start the execution using the starter. We will then understand how to poll for decision tasks and do the orchestration. Eventually, we will also see how to poll for activity tasks to perform the scheduled tasks.

There are few other terminologies you can do well to understand before we get into the details on practical application. Getting to know these terms can ease the process of understanding the remaining concepts discussed in this chapter:

- **Workflow** – Workflow is a combination of one or more processing steps that can be executed on multiple computing devices.
- **Workflow type** – Combination of domain, workflow name, and version to uniquely identify a workflow.
- **Workflow execution** – An instance of orchestration for specific data. For example, an order made by a customer can trigger one execution. Each execution is identified by domain, workflow id, and run id.
- **Activity** – A step in a workflow that can be executed by a worker. It is also called a task.
- **Activity type** – Unique combination of domain, activity name, and version.

- **Task list** – A logical grouping of activities/tasks that are part of a workflow. Task lists are used to poll for activities. They provide a mechanism to route tasks to workers.
- **Domain** – A logical grouping of all the components of a workflow, such as activities.

Creating Types

You may already be familiar with the pattern of learning here. We will first understand how to do it through the AWS console and slowly graduate to doing it programmatically. Setting up a workflow is a multistep process; we have to first create a domain to encapsulate our workflow-related entities. We have to register a new workflow type under that domain and then create activity types that need to be part of the same domain.

Register Domain

Console Method

Creating a workflow from the console is simple enough, we just click a few links and provide all the details it asks us for. However, certain aspects cannot be created using the console alone. Implementations for the activity workers and deciders cannot be supplied while creating workflow through the console. We will just create entities with metadata and mandatory parameters such as timeouts. We will cover implementations in subsequent sections.

To create a domain and its entities, open the SWF service from the AWS Management Console. The service home page should look like in Figure 4-2. It is organized similar to other services. The left-hand navigation is to switch to different views of the SWF service. The remaining portions give access to various functionalities of SWF such as creating workflow types and activity types and finding workflow executions.

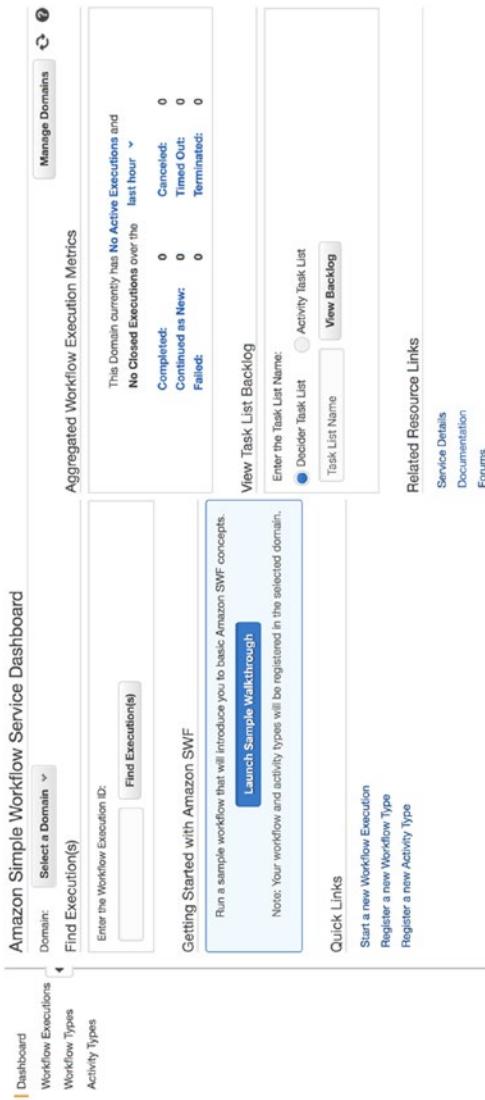


Figure 4-2. SWF Service Dashboard

To create a workflow type, we first need to create a domain. Click the **Manage Domains** button located in the top-right corner of the Dashboard page. This opens up a wizard that shows the domains we have created, and you should find a Register New button to create a new one. Click it to open the wizard, as shown in Figure 4-3.

The screenshot shows a web-based registration form titled 'Register Domain'. At the top right is a 'Cancel' button. Below the title, a sub-instruction says 'Provide the details of your new Domain below, then click Register'. The form contains three main input fields: 'Name:' with a value of 'RestaurantDomain', 'Workflow Execution Retention Period:' with a value of '7' followed by 'Days', and 'Description:' which is empty. At the bottom right is a large 'Register' button.

Figure 4-3. Register Domain Wizard

You can provide the name of the domain. It needs to be unique in a region in an AWS account. Give a meaningful name such as RestaurantDomain or OrderDomain. The next input in the wizard is to specify the duration for which the execution must be retained by SWF. Once the period has passed, the workflow executions will be removed and can no longer be queried or listed. The final input, Description, is optional and self-explanatory. Finally, click the Register button to let the domain be created. It should now show up in the domains list.

CLI Method

To create the same domain from the command line, we can use the `register-domain` sub-command. The command we can use with AWS for all Simple Workflow Service-related tasks is `SWF`. As you may remember, each AWS command takes the following format:

```
$ aws [options] <command> <sub-command> [parameters]
```

From the console method, you may have noticed that we need to pass name, retention period, and, optionally, description. Converting that into an actual command, an example command to create an exact domain will be as follows. As the description is optional, you can skip it in the command as well:

```
register-domain  
--name RestaurantDomain  
--description 'The domain for managing orders'  
--workflow-execution-retention-period-in-days 7
```

You should have already configured the user credentials to be the default profile. Otherwise, you will have to provide the --profile parameter and give the name of the profile you want to use for registering the domain. In any case, the user should also have access to do so. If the user doesn't have access, go to IAM and give full access to the user.

If all goes well, you should not see any message as output to the command. You can immediately go to the AWS console to find it in the domain dropdown. If you are CLI savvy, you can also execute the following command to get a list of domains from the command line:

```
$ aws swf list-domains --registration-status REGISTERED
```

The parameter registration status is mandatory, and it accepts either REGISTERED or DEPRECATED. We occasionally deprecate the domains when we no longer use them, so we can filter them out by just asking for domains with REGISTERED status. The output is a JSON with all the domains we registered and not deprecated yet, as shown below. Your list may vary:

```
{  
    "domainInfos": [  
        {  
            "status": "REGISTERED",  
            "name": "ExampleDomain"  
        },  
        {  
            "status": "REGISTERED",  
            "name": "RestaurantDomain",  
            "description": ""  
        },  
        {  
            "status": "REGISTERED",  
            "name": "order_domain"  
        },  
        {  
            "status": "REGISTERED",  
            "name": "order_domain_1",  
            "description": "the orders workflow domain"  
        },  
        {  
            "status": "REGISTERED",  
            "name": "test-01"  
        }  
    ]  
}
```

API Method

I hope you have installed aws-sdk already when you worked on previous chapters. With that assumption, I'm going to jump straight ahead to the part where we register the domain (Listing 4-1). I'm also skipping the portion of code that gets the credentials and configures them so that the code can get access to the AWS services.

Listing 4-1. javascript-sources/chapter-4/demo-01/register-domain.js

```
let swf = new AWS.SWF();  
  
swf.registerDomain({  
    name: 'Order_domain_2',  
    workflowExecutionRetentionPeriodInDays: '10'  
}).promise()  
    .then(response => console.log(response))  
    .catch(err => console.log(err));
```

This snippet first creates an SWF client from the AWS object we create by requiring aws-sdk. The client provides a method registerDomain to which we can pass the params. The response will be empty if there are no errors, just like the CLI command. The possible error can be Domain Already Exists exception or Access Denied exception.

The equivalent Java snippet of the code will be very much similar except for the request params being passed as a RegisterDomainRequest object, as shown in Listing 4-2. The SWF client is created using the builder class provided by SDK, AmazonSimpleWorkflowClientBuilder. The parameters can be set using getter methods of the object. Once the domain is created, you can query using the method listDomains.

Listing 4-2. java-sources/chapter-4/demo-01/RegisterDomain.java

```
AmazonSimpleWorkflow simpleWorkflow =  
AmazonSimpleWorkflowClientBuilder.defaultClient();  
RegisterDomainRequest request = new RegisterDomainRequest();  
request.setName("OrdersDomain_3");  
request.setWorkflowExecutionRetentionPeriodInDays("7");  
simpleWorkflow.registerDomain(request);
```

With this experience, try to find out how to do it in other programming languages like Python. It would be very similar, except for a few syntactical differences.

Register Workflow Type

Once the domain has been created, we then need to create a workflow type. A workflow takes slightly more parameters than a domain. Unlike a domain, a workflow type also consists of a version along with a name and a defaultTaskList as mandatory parameters. A few other timeouts can also be provided. Let us get us into the details.

Console Method

Go back to the Dashboard and select the domain from the dropdown with label Select a Domain. By choosing a domain, we can create corresponding workflow types and activity types in that domain. Once selected, click the Register a New Workflow Type link under the Quick Links section of the Dashboard page. This brings up a wizard to enter a few mandatory details, as shown in Figure 4-4.

Register New Workflow

Provide the details of your new Workflow below:

Domain*: RestaurantDomain

Workflow Type Name*: OrderWorkflow

Workflow Type Version*: 0.1

Default Task List*: OrderTasks

Default Execution Start to Close Timeout*: 12 hours

Default Task Start to Close Timeout*: 30 minutes

Continue

The screenshot shows a registration form for a new workflow. At the top, it says 'Register New Workflow' and has a 'Cancel' button. Below that, it asks for details of the new workflow. It includes fields for 'Domain*' (set to 'RestaurantDomain'), 'Workflow Type Name*' (set to 'OrderWorkflow'), 'Workflow Type Version*' (set to '0.1'), 'Default Task List*' (set to 'OrderTasks'), and two timeout settings: 'Default Execution Start to Close Timeout*' (set to '12 hours') and 'Default Task Start to Close Timeout*' (set to '30 minutes'). At the bottom right is a 'Continue' button.

Figure 4-4. Register Workflow Wizard

As you can guess, the name and version are to identify a specific workflow type. In SWF, once you create a workflow, you cannot modify it without changing the version; in other words, they are immutable. You can access previous versions of a workflow and start execution even after you create a new version. The task list is a logical grouping of tasks so that they can be polled for easily. You can provide it as OrderTasks. We will create activities and add them to this task list in subsequent steps.

The other two parameters in this page are timeouts. The Default Execution Start to Close Timeout is to indicate how long should each execution of workflow be. If a customer clicks Order on the UI, the workflow execution should start. If we are waiting on a specific vegetable to be cut for the dish, prep task is still underway, which means the workflow is still getting executed. Once we reach 12 hours, there is no point in trying to keep executing the workflow; we can cancel the order and terminate workflow execution.

The other timeout is for individual activities in a workflow. If a task is not finished in the given timeframe, it will be considered as a failed attempt and be retried. For both the timeouts, you can change the time

units among seconds, minutes, hours, and days. Make sure to give enough timeout for the entire workflow. It should be more than the cumulative time it takes for all the tasks in that workflow. Once you enter all the information, you can click the Continue button to navigate to the next page of the wizard where you have to provide Default Task Priority, Description, and Default Child Policy and Default Lambda Role. These are optional, and we will cover some of them when the time is right. Let us keep it straight for now and finish creating the workflow type.

Click the **Review** button to move to the final page where you can find all the information you have provided in the previous pages of the wizard and can click the button **Register Workflow** to complete creating the workflow.

Assuming that you have provided all the details correctly, you should find the workflow listed in the Workflow Types page of SWF. Just click **Workflow Types** in the left-hand navigation bar to switch to the page. If you can't see the workflow type you created, make sure that you are in the right domain. You have a similar dropdown to switch the domain at the top of the page. You can also filter the workflow types by name if there are many workflows and versions. In this page, you can also deprecate the specific version of a workflow and can register a new one.

CLI Method

The command is swf as usual, and the sub-command is register-workflow-type. The command takes the following form. I have omitted the usual generate-cli-skeleton and cli-input-json parameters of the command as they work the same way as with every other command:

```
register-workflow-type
  --domain <value>
  --name <value>
  --workflow-version <value>
  [--description <value>]
```

```
[--default-task-start-to-close-timeout <value>]
[--default-execution-start-to-close-timeout <value>]
[--default-task-list <value>]
[--default-task-priority <value>]
[--default-child-policy <value>]
[--default-lambda-role <value>]
```

The parameters correspond to the same fields I have explained in the previous section when we are creating using UI, so I'm not going to explain them again here. The domain, name, and workflow version uniquely identify a workflow type. The workflow version can be any string and need not be a number.

The parameters enclosed in brackets, such as `description`, `default-task-start-to-close-timeout`, and others, are optional. However, if you don't provide them while creating a workflow type, you will have to provide them when the workflow is being executed. Here is an example of the command. Notice the format of the parameter `default task list`; we need to pass a JSON with the parameter name:

```
$ aws swf register-workflow-type --domain order_domain
--name order_flow --workflow-version seven --default-task-
list='{"name": "order_steps"}' --default-task-start-to-close-
-timeout 9
```

API Method

The following JavaScript snippet, as shown in Listing 4-3, creates a workflow type. Notice the `defaultTaskList` JSON following the same rules as the CLI command. Another observation you can make is that the timeout parameters are in number of seconds, but we have to pass them as strings.

Listing 4-3. javascript-sources/chapter-4/demo-01/register-workflow-type.js

```
const params = {
    domain: "Order_domain_2",
    name: "order_workflow",
    version: "0.5",
    defaultTaskList: {
        name: "OrderSteps"
    },
    defaultTaskStartToCloseTimeout: "300",
    defaultExecutionStartToCloseTimeout: "1800"
};

swf.registerWorkflowType(params);
```

Remember that this snippet ignores the handling of promise, as it is similar to every other piece of snippet. If the workflow type already exists or if any other occurs, the promise fails, and the catch scenario comes into play if written. You can find the complete code if you open the source file from the downloaded code.

Register Activity Types

Activities or tasks are small chunks of work carried out by workers. They can be automated activities such as running a lambda function, or humans may have to perform some tasks to complete the activity physically. A drone may have to deliver the packed order to the customer's home physically. Irrespective of what that task may accomplish, we need to first register it under a specific task list so that it can be pulled from SWF.

Console Method

Go back to the Dashboard page of SWF and click the **Register a New Activity Type** hyperlink in the Quick Links section. Alternatively, you can click the Activity Types link on the left navigation bar to switch to the activities page where you will find a familiar **Register New** button. Either way, you should see a wizard asking for the details necessary for an activity to be registered split into two pages, as shown in Figure 4-5.

1

Register New Activity

Provide the details of your new Activity below:

| | | |
|--------------------------------|------------------|---------|
| Domain* | RestaurantDomain | |
| Activity Type Name* | Ship | |
| Activity Type Version* | 0.1 | |
| Task List | OrderTasks | |
| Task Priority | | |
| Task Schedule to Start Timeout | 05 | minutes |
| Task Start to Close Timeout | 30 | minutes |

Continue

2

Register New Activity

Provide additional options for your new Activity below:

| | |
|--------------------------------|---------|
| Description | |
| Heartbeat Timeout | seconds |
| Task Schedule to Close Timeout | seconds |

Back **Review**

Figure 4-5. Registering Activity

The first page asks for **Activity Type Name** and **Activity Type Version**, which are mandatory details for each task we created. The next parameter Task List is for grouping multiple tasks in a task list. **Task Priority** is to decide which activity should be executed first if there is a conflict. The **Task Schedule to Start Timeout** is the duration after which a task is considered as timed out if a worker doesn't acknowledge that it has started working on a task.

The final timeout in the first page of the wizard, **Task Start to Close Timeout**, is the total duration to complete a task. Each task may have different timeouts depending on the amount of time it takes for that activity to be completed in the real world and some buffer time. Always plan to give a buffer time so that they won't inadvertently be timed out when they are just about to be finished.

After filling out all the details, click Continue, and you will see a follow-up wizard asking for additional details. You can describe the activity task in detail in the **Description** tab. **Heartbeat Timeout** is the maximum time, by default, before a worker that is processing a task of this type reports progress by calling RecordActivityTaskHeartbeat. The **Task Schedule to Close Timeout** is the total maximum time for which an activity task can be scheduled to perform. However, note that these details are not mandatory fields. Click the Review button to review all the details filled till now and simply click Register Activity. You can now see that the activity is now successfully registered and is visible in the Activity Types table right away.

CLI Method

The CLI command for registering activity types takes the following format. The domain, name, and activity versions are mandatory parameters, while the rest of the parameters are optional while registering an activity:

```
register-activity-type
  --domain <value>
  --name <value>
  --activity-version <value>
  [--description <value>]
  [--default-task-start-to-close-timeout <value>]
  [--default-task-heartbeat-timeout <value>]
  [--default-task-list <value>]
  [--default-task-priority <value>]
  [--default-task-schedule-to-start-timeout <value>]
  [--default-task-schedule-to-close-timeout <value>]
```

Even though the timeouts and task list are optional while registering the activity type, they are necessary during the execution of the workflow. We either have to provide the default timeouts during registering them or we have to provide them while scheduling an activity of this type. The following is an example of the command to register an activity type:

```
$ aws swf register-activity-type --domain order_domain  
--name prep_task --activity-version v2.2  
--default-task-start-to-close-timeout 1800  
--default-task-heartbeat-timeout 600  
--default-task-schedule-to-start-timeout 600  
--default-task-schedule-to-close-timeout 2100
```

API Method

The method that performs an equivalent task of registering an activity type is `registerActivityType` in both Java and JavaScript. The Python method is `register_activity_type` which takes keyword arguments. Just for a change, in this snippet, as shown in Listing 4-4, I'm going to show the Python version of the code.

Listing 4-4. `python-sources/chapter-4/demo-01/register-activity-type.py`

```
client = boto3.client('swf')

response = client.register_activity_type(
    domain='Order_domain_2',
    name='prep_task_2',
    version='v2',
    defaultTaskStartToCloseTimeout='1800',
    defaultTaskHeartbeatTimeout='300',
    defaultTaskList={
        'name': 'order_tasks'
    },
```

```
defaultTaskScheduleToStartTimeout='300',
defaultTaskScheduleToCloseTimeout='2100'
)
print response
```

We import the boto3 library and create a client for SWF. We can invoke register_activity_type on the SWF client. The function call is a blocking type, just like Java, so we don't need to implement any callback or handle promises. It returns a JSON object with HttpStatusCode of 200:

```
{
    "ResponseMetadata": {
        "RetryAttempts": 0,
        "HTTPStatusCode": 200,
        "RequestId": "fc01b008-5a34-445e-aaf9-dcb2ccda3eb5",
        "HTTPHeaders": {
            "x-amzn-requestid": "fc01b008-5a34-445e-aaf9-
                dcb2ccda3eb5",
            "date": "Sun, 16 Jun 2019 04:50:13 GMT",
            "content-length": "0",
            "content-type": "application/x-amz-json-1.0"
        }
    }
}
```

With this, we have completed the necessary type registrations for us to proceed to the part of implementation of the workers to execute the workflow.

Workflow Execution

You may be tempted to click Start New Workflow Execution in the Quick Links section of the Dashboard. You can, and the workflow execution will start alright, but it won't progress any further, because we need to implement deciders and activity workers. It waits for 12 hours or whatever timeout you set for the parameter Default Start to Close Timeout.

To observe it in action, click the button to start a workflow execution. You will be prompted to provide the workflow name and version in the right domain. I have created a domain named RestaurantDomain with the workflow name OrderWorkflow and the version 0.1. You can enter them in the wizard. Alternatively, you can navigate to the Workflow Types view and select the workflow and click the Start New Execution button. This will bring up the same pop-up to start execution, and you will have to enter a run id. You can enter any text to differentiate each execution. One idea is to use a combination of customer id and order time we have used to identify each order in the orders table uniquely.

After filling the required fields, the wizard would look as shown in Figure 4-6. If you click the hyperlink Advanced Options, you will be able to override some properties you have set earlier, such as task list, timeouts, and a couple of others.

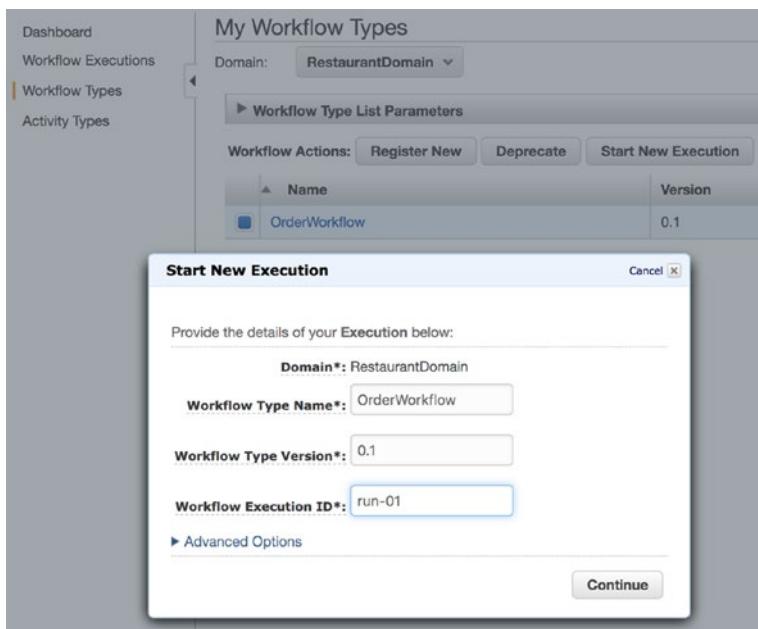


Figure 4-6. Starting New Execution

You can try changing the fields, but I'm going to skip them for now. Click the Continue button to move to the next page of the start execution wizard. The next page contains just one field, Input. We can use this input to pass some state information from one task to another, or we can read the state from DynamoDB. Let us skip it for now.

The second page has a different set of advanced options for us to fiddle with. Ignore them likewise and click Review to move to a page that displays the information we have provided in this wizard. Click the Start Execution button to complete the formality of starting the execution. You can immediately find the workflow execution in the Workflow Executions page.

Click the workflow execution in the table. This opens the summary of the workflow execution. Click the Events tab to see what has been going on in the workflow as shown in Figure 4-7. Notice that there are two events listed in the Events tab right now.

The first event is the workflow execution started event. It is always going to be the first activity of any workflow execution. The second event is `DecisionTaskScheduled`. This means if a decider polls for decision tasks when a decision task is scheduled, it gets a response. Otherwise, the decider gets an empty response. The decider task repeats this polling on a regular basis. Once the decider analyzes the list of history events and finds that this is the right time for a task to be executed, it schedules an activity task.

The screenshot shows the AWS Simple Workflow Service console. On the left, there is a navigation sidebar with options: Dashboard, Workflow Executions (which is selected and highlighted in orange), Workflow Types, and Activity Types. The main area is titled "Workflow Execution: run-01". Below the title, it says "Domain: RestaurantDomain". There are four tabs at the top of the main content area: Summary, Events (which is selected and highlighted in blue), Activities, and Child Workflows. The "Events" tab displays a table of events. The table has columns: Event Date, ID, and Event Type. There are two rows of data:

| Event Date | ID | Event Type |
|----------------------------------|----|--------------------------|
| Sun Jun 16 11:19:41 GMT-700 2019 | 2 | DecisionTaskScheduled |
| Sun Jun 16 11:19:41 GMT-700 2019 | 1 | WorkflowExecutionStarted |

Figure 4-7. Workflow Execution Events

You should then see `DecisionTaskStarted`, `DecisionTaskCompleted`, and `ActivityTaskScheduled` as the next set of events in the Events tab of the same workflow execution. The events list may also have other types of events such as `ActivityTaskTimedOut`. It all depends on what happened when the workflow is being executed. Our execution is going to sit like a duck as we don't have any running deciders or workers. Let us look at how to do that.

Implementing Actors

For a workflow to execute smoothly from beginning to end, we need to implement actors to do the work. There are mainly three types of actors that can exist in a workflow; they drive the workflow through various stages. They are as follows:

- Decider – This is an orchestration actor of the workflow. The responsibility of a decider is to regularly poll for the history of events and deduce what is to be done next and schedule the activity.
- Activity worker – This actor performs the critical functionality necessary for the business to operate. You can correlate this to the Lambda SQS worker we created in Chapter 2. In that chapter, we read the message from queues and processed the order. Activity workers regularly poll for any scheduled activities and work on them if there's any.
- Starter – The whole responsibility of this actor is to kick-start the workflow execution. Once the workflow is successfully kick-started, the job of the starter actor is over.

We can implement the workers using any type of compute. It can be a Lambda function or a Python process that runs in a Docker container running on ECS/Fargate. It can just be a program running in our machine. We can have a workflow with actors implemented in a different set of computes. In this section, we will implement different kinds of actors using different forms of compute. We will ignore the CLI mechanism as it really doesn't do much for us in the real world.

There is another actor that is not discussed often; we can also have a terminator, not the one with Arnold Schwarzenegger, but a software terminator that can close flawed executions. If you remember, we have earlier cancelled some orders when the customer clicks the Order button for the second time. In those instances, we don't really want to keep the workflow running. However, we can also terminate an execution through a decision task, so we don't need this actor as often as the other ones.

Implementing Starter

We have executed the workflow from the AWS console, but how can we do that programmatically? Just like how we have created the types. We can use the same SWF client for starting a workflow. The API for initiating a workflow is `startWorkflowExecution`. In Python, it would follow a snake case naming convention, so the method name is going to be `start_workflow_execution`. In either case, a request for starting a workflow has the following parameters:

- domain – The name of the domain in which the workflow is created.
- workflowId – The user-defined identifier associated with the workflow execution.
- workflowType – Name and version of the workflow.
- taskList – The task list to use for the decision tasks generated for this workflow execution. This overrides the default value specified when registering the workflow type.
- taskPriority – The task priority to use for this workflow execution. This overrides any default priority that was assigned when the workflow type was registered.

- input – The input for the workflow execution. This is a free-form string which is meaningful to the workflow execution, for example, customer id and order time.
- executionStartToCloseTimeout – The total duration for this workflow execution. This overrides the default value specified when registering the workflow type.
- tagList – The list of a maximum of five tags to associate with the workflow execution.
- taskStartToCloseTimeout – Maximum duration of each decision task for this execution.
- childPolicy – If set, specifies the policy to use for the child workflow executions of this workflow execution if it is terminated.
- lambdaRole – The IAM role to attach to this workflow execution. Executions of this workflow type need IAM roles to invoke Lambda functions.

Many of these parameters are optional. Some of them become mandatory if we don't provide them while creating a workflow. The parameters task list and timeouts are mandatory if not provided while creating the workflow; if we provide them during start execution, the values will be overridden. Input is a string in any format that works for our application. Taglist is for us to organize the executions for filtering and maintenance. The following is a sample code for starting a workflow execution in JavaScript SDK:

```
let params = {
  domain: "RestaurantDomain",
  workflowId: "prog-run-01",
  workflowType: {
    name: "OrderWorkflow",
    version: "0.1"
}
```

```
    } ,
  taskList: {
    name: "OrderTasks"
  },
  input: "OrderID",
  executionStartToCloseTimeout: "300",
  taskStartToCloseTimeout: "60"
};

swf.startWorkflowExecution(params);
```

As a response to the start workflow execution API, we receive a run id which we can ignore unless we want to keep track of a workflow programmatically for whatever reason. Usually, we don't have to worry about it, because the deciders and workers take care of the further execution of the workflow.

As we have seen how to start a workflow, it is now time for us to integrate into our application. Our application in the previous chapters is triggering a lambda function that saves the record in DynamoDB and inserts a message in a queue. Now, as we don't have to maintain multiple queues anymore, let us change the lambda function code to start a workflow. The following is the source code for the lambda handler:

```
const AWS = require("aws-sdk");

exports.handler = async (event, context) => {
  let ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
  let swf = new AWS.SWF();

  let ts = new Date().toISOString().replace("T", " ")
    .replace("Z", "");
  // ConstructDynamoDB putItemRequest
  try {
    await ddb.putItem(putItemRequest).promise();
```

```
let workflowParams = {
    domain: "RestaurantDomain",
    workflowId: event.user_id + "_" + ts,
    workflowType: {
        name: "OrderWorkflow",
        version: "0.1"
    },
    input: JSON.stringify({
        user_id : event.user_id,
        order_time: ts
    })
};

await swf.startWorkflowExecution(workflowParams).promise();
console.log("Successfully inserted data and started
workflow");
context.succeed(ts);
} catch (err) {
    context.fail(err);
}
};

});
```

Notice that I have not passed the task list and timeouts as they are created when the workflow is being created. There's no need to pass them unless we need to override them. We have just passed a minimum amount of data required for the workflow to be executed. We have also passed user id and order time as the input of the workflow. This way, the decider and activity workers can query the order from the database.

You can either modify the existing save-order lambda or create a new one, but make sure you construct 'putItemRequest' object based on the previous examples. You need to ensure that the role of the lambda function has access to trigger SWF workflows as well. You can just open the role in

IAM and attach the predefined policy SimpleWorkflowFullAccess. If you modified the existing lambda function, you don't have to modify your application's `lambda_caller.js`; otherwise, change the name of the lambda function used in the save order function.

Alternatively, we can also trigger the workflow from JavaScript of the web page. However, there is a bug in the deployment process that is not pushing the `swf.js` file at the time of writing this book, so you may receive an error if you try that. Also, your web page has to take the responsibility of inserting data in DynamoDB, so to reduce the friction, let us take the easy route of updating the lambda function to start the workflow instead of inserting the message in the queue.

Now, run the application, as usual, using `npm run serve`, and click the Order button. If there is any mistake, you can see the errors in the Console tab of developer tools in your browser. If everything goes as planned, you should see that a new workflow execution has been started, as shown in Figure 4-8.

| Workflow Execution ID | Run ID | Name (Version) |
|--|------------|---------------------|
| eef94240-953c-11e9-8cdd-79d7b0b457b6_2019-06-22 22:28:04.611 | 23R6akyusw | OrderWorkflow (0.1) |
| eef94240-953c-11e9-8cdd-79d7b0b457b6_2019-06-22 22:27:47.117 | 23Sj+PYVDG | OrderWorkflow (0.1) |

Figure 4-8. Executions Initiated by Clicking the Order Button

If you notice, there are two executions, which means I ordered two items from our restaurant's home page. Each execution id is the combination of user id and timestamp. As we anyways have embedded these fields in execution id, we really didn't need to pass them in input again. All we need to do is to split the execution id by an underscore;

we have the same information available from parsing input. In any case, let us take the workflow forward by implementing other actors of workflow execution.

Implementing Decider

Every time a state change occurs, SWF schedules a decision task as you might have already observed. A decider, once started, has to perform three activities:

- Poll for decision tasks on regular intervals.
- Interpret the workflow execution history and make decisions.
- Respond with decisions.

Polling for Decision Tasks

In JavaScript SDK, we can poll for decision tasks using the method `pollForDecisionTask`, which takes an object of type `PollForDecisionTaskInput`. The following is the sample code snippet for the same request:

```
let input = {
    domain: "RestaurantDomain",
    taskList: {
        name: "OrderTasks"
    },
    reverseOrder: true
};

swf.pollForDecisionTask(input).promise()
    .then(data => console.log(JSON.stringify(data, null, 2)))
    .catch(err => console.log(err));
```

The preceding code asks SWF to return one decision task associated with task list OrderTasks. Remember that there is no way to poll for a specific workflow execution or even workflow. The only available granularity is taskList. If you have multiple workflow executions of the same workflow or if the same task list is shared across multiple workflows, any one decision task is returned. The response will be similar to this:

```
{  
  "taskToken": "AAAAKgAAAAIAAAAAAAAAAzpHM72FC... +",  
  "startedEventId": 3,  
  "workflowExecution": {  
    "workflowId": "eef94240-953c-11e9-8cdd-  
                  79d7b0b457b6_2019-06-23 18:12:49.769",  
    "runId": "23EsSaZs4jhYnPkzXb8I9pSWY8KC2frMqbqnoQEJmg3F8="  
  },  
  "workflowType": {  
    "name": "OrderWorkflow",  
    "version": "0.1"  
  },  
  "events": [  
    {  
      "eventTimestamp": "2019-06-23T18:13:20.597Z",  
      "eventType": "DecisionTaskStarted",  
      "eventId": 3,  
      "decisionTaskStartedEventAttributes": {  
        "scheduledEventId": 2  
      }  
    },  
    {  
      "eventTimestamp": "2019-06-23T18:12:50.658Z",  
      "eventType": "DecisionTaskScheduled",  
      "eventId": 2,  
    }]
```

```
"decisionTaskScheduledEventAttributes": {  
    "taskList": {  
        "name": "OrderTasks"  
    },  
    "startToCloseTimeout": "1800"  
}  
},  
{  
    "eventTimestamp": "2019-06-23T18:12:50.658Z",  
    "eventType": "WorkflowExecutionStarted",  
    "eventId": 1,  
    "workflowExecutionStartedEventAttributes": {  
        "input": "{\"user_id\":\"eef9...457b6\", \"order_time\":  
            \"2019-06-23 18:12:49.769\"}",  
        "executionStartToCloseTimeout": "43200",  
        "taskStartToCloseTimeout": "1800",  
        "childPolicy": "TERMINATE",  
        "taskList": {  
            "name": "OrderTasks"  
        },  
        "workflowType": {  
            "name": "OrderWorkflow",  
            "version": "0.1"  
        },  
        "parentInitiatedEventId": 0  
    }  
},  
]  
,"previousStartedEventId": 0  
}
```

CHAPTER 4 SIMPLE WORKFLOW SERVICE

If we simplify the response to understand it better, the response looks like the following schema. It has a taskToken, using which we can respond with our decisions and SWF will attach the decisions to the correct execution of the workflow:

```
{  
    "taskToken": "",  
    "startedEventId": ,  
    "workflowExecution": {  
        "workflowId": "run-02",  
        "runId": ""  
    },  
    WorkFlowName & Version  
    "events": [  
        DecisionTaskStartedEvent  
        DecisionTaskScheduledEvent  
        WorkflowExecutionStartedEvent  
    ],  
    "previousStartedEventId": 0  
}
```

The other parameters in this JSON are to indicate what is the workflow execution id and what events occurred so far. If you have observed our request, we have asked SWF to return events in reverse order, by setting the parameter `reverseOrder` to true while polling for decision tasks. By default, it would return them in the chronological order in which they occurred.

Each event also contains attributes of their own, even though the above simplified schema doesn't show them. For example, the workflow started event contains attributes such as input we pass while starting the workflow. Let us first review a bit more about `pollForDecisionTask`. The input of `pollForDecisionTask` API contains the following parameters:

- domain – The name of the domain containing the task lists to poll.
- taskList – Specifies the task list to poll for decision tasks.
- identity – Some identifier of the decider. As there can be more than one decider in a cloud environment, it is helpful to keep track of which decider is polling and scheduling tasks.
- maximumPageSize – The maximum number of events to be returned per page.
- nextPageToken – If there is more than one page worth of events, we also get the next page token in the response. It can be used to fetch the next set of events.
- reverseOrder – Usually, the events are returned in chronological order. However, we rarely need it as we often have to just look at the past few events to decide what to do next. We can request SWF to return the events in reverse order by setting this property to true.

With the knowledge of what can be done, now it is time for us to take the example further. With the same input or domain and task list and reverseOrder set to true, let us see how to extract the order information to make important decisions:

```
swf.pollForDecisionTask(input).promise()
  .then(data => {
    if(data.workflowExecution) {
      let parts = data.workflowExecution.workflowId.
        split("_");
      let userId = parts[0]
      let order_time = parts[1]
      handleOrder(userId, order_time, data.taskToken);
```

```
    } else {
        console.log('Empty response when polling for
decision tasks')
    }
})
.catch(err => console.log(err));
```

When the promise is returned, we check if we have an empty response. Usually, the decider will be long polling SWF for 60 seconds. If no decision tasks are scheduled within that time, SWF returns an empty response. Hence, we are making sure that it is not empty before we try to split the workflow execution id by an underscore. Otherwise, we will get an exception.

As we have earlier decided to combine user_id and order_time using an underscore, we just extracted them using the same mechanism. We will see how to make decisions based on the order state now.

Making Decisions

Our code so far just fetches the events but doesn't do anything useful so far. We need to implement logic to direct workflow execution. In our usual workflow example, we usually fetch the order status, and if the order status is CONFIRMED, we set the order to PREP_QUEUE at which point the preparation activity worker comes in to play and modifies the order status to PREPARED.

We can go about making a decision using several approaches. One is to go through the event history and decide what happens. Our use case is much simpler; we can just fetch the order state from DynamoDB. In the previous section, we have offloaded the responsibility to the handleOrder method and passed it the information necessary to fetch the order state and task token. The task token is going to be necessary for sending back the response to SWF about the decision the decider has made.

Fetching the order state is familiar to you. We create a DynamoDB client and invoke the query method and pass the key condition expression to query the exact order. The following snippet does that:

```
function handleOrder(userId, order_time, task_token) {
    let docClient = new AWS.DynamoDB.DocumentClient();
    let query_input = {
        TableName : "orders",
        KeyConditionExpression: "user_id = :id and
        order_time = :ts",
        ExpressionAttributeValues: {
            ":id":userId,
            ":ts":order_time
        }
    };
    docClient.query(query_input).promise()
        .then(data => orchestrate(data.Items[0], task_token))
        .catch(err => console.log(err));
}
```

Once we have the latest order state, we invoke the orchestrate method. In the orchestrate method, if the order status is CONFIRMED, we need to send it to preparation, so we create a decision that tells SWF to start prep activity or schedule a prep activity to be precise. The following is the decision object that can be used to tell SWF to schedule an activity of type Prep:

```
decision = {
    decisionType: "ScheduleActivityTask",
    scheduleActivityTaskDecisionAttributes: {
        activityId: "1234",
        taskList: {
```

```

        name: "PREP_LIST"
    },
    activityType: {
        name: "Prep",
        version: "0.1"
    },
    input: JSON.stringify({
        user_id: order.user_id,
        order_time: order.order_time
    })
}
};


```

The decision type indicates what should happen. One such type is ScheduleActivityTask that is used to inform SWF to schedule an activity of the given type. Various other types are listed in Table 4-1.

Table 4-1. *Various types of decisions a decider can make*

| Decision Type | Use Case |
|---------------------------|---|
| ScheduleActivityTask | To start an activity. |
| RequestCancelActivityTask | To cancel an activity that has already been started. |
| CompleteWorkflowExecution | To mark the workflow execution as complete. |
| FailWorkflowExecution | To mark the workflow execution as finished and fail it. |
| CancelWorkflowExecution | To cancel the workflow execution currently running. |

(continued)

Table 4-1. (continued)

| Decision Type | Use Case |
|---------------------------------|--|
| ContinueAsNewWorkflowExecution | Closes the current workflow execution and restarts it with a new runId. Workflow execution id stays the same. |
| RecordMarker | Marker is a special event in the workflow history to help deciders to skip the remaining history of events. This can have meta information to mark progress. |
| StartTimer | Starts a timer for this workflow execution and triggers a TimerStartedEvent. When the timer runs out, it records a TimerFired event. |
| CancelTimer | Cancels the previously started timer and records a TimerCancelled event. |
| SignalExternalWorkflowExecution | Requests a signal to be delivered to external workflow execution. |
| RequestCancel | Initiates a request to cancel an external workflow run. |
| ExternalWorkflowExecution | |
| StartChildWorkflowExecution | Starts a child workflow execution. |
| ScheduleLambdaFunction | Schedules a lambda function to be executed. |

The decision type determines which attributes have to be passed. For a ScheduleActivityTask, we need to pass ScheduleActivityTaskDecision Attributes, and if the type is RecordMarker, we have to pass the RecordDecisionMarkerAttributes object. Each object has a different set of attributes to be passed.

In our example, we have passed ScheduleActivityTaskDecisionAttributes where other parameters indicate which activity to be scheduled and under which task list the activity should be listed and what input needs to be passed to the activity. However, if we have a default task list configured for the activity while creating the type, we don't have to pass the task list. We can also pass an activity id and input. We can pass input so that the activity can skip trying to fetch the state from DynamoDB. For now, we can just use the state fetched from DynamoDB. The following code generates the correct decision based on current order status:

```
if (order.order_status === 'Confirmed') {  
    decision = generateDecision("Prep");  
} else if (order.order_status === 'PREPARED') {  
    decision = generateDecision("Cook");  
} else if (order.order_status === 'COOKED') {  
    decision = generateDecision("Pack");  
} else if (order.order_status === 'PACKED') {  
    decision = generateDecision("Ship");  
}
```

Once the decision is generated, we can invoke respondDecisionTaskCompleted. We need to pass taskToken so that SWF can attach the activity schedule to the correct workflow execution. Each response can include more than one decision, so we need to pass an array of decisions, even if there is just one decision. The following code does just that:

```
swf.respondDecisionTaskCompleted({  
    taskToken: task_token,  
    decisions: [decision]  
}).promise().then(console.log('Activity task scheduled'));
```

We need to complete the decision task before the configured timeout; otherwise, the decision task is marked as timed out, and the next decider is scheduled to pick up the same decision task again. The same cycle of polling for decisions and generating a decision gets repeated. Here is the complete code, as shown in Listing 4-5.

Listing 4-5. javascript-sources/chapter-4/demo-03/decider.js

```
const AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.SharedIniFileCredentials
    ({profile: 'admin'});

let swf = new AWS.SWF();

let input = {
    domain: "RestaurantDomain",
    taskList: {
        name: "OrderTasks"
    },
    reverseOrder: true
};

function orchestrate(order, task_token) {
    let decision = {};

    function generateDecision(activityName) {
        return {
            decisionType: "ScheduleActivityTask",
            scheduleActivityTaskDecisionAttributes: {
                activityId: activityName + "_" + Math.random(),
                activityType: {
                    name: activityName,
                    version: "0.3"
                }
            }
        };
    }

    let decisions = [];

    for (let i = 0; i < order.length; i++) {
        let item = order[i];
        let decisionData = generateDecision(item.activity);
        decisionData.name = item.activity;
        decisionData.id = item.id;
        decisionData.order = i;
        decisions.push(decisionData);
    }

    let decisionString = JSON.stringify(decision);
    let decisionsString = JSON.stringify(decisions);

    let params = {
        decisionString: decisionString,
        decisionsString: decisionsString,
        taskToken: task_token
    };

    swf.respondDecisionTaskCompleted(params, function(err, data) {
        if (err) {
            console.log("Error: " + err);
        } else {
            console.log("Success: " + data);
        }
    });
}
```

```
        },
        input: JSON.stringify({
            user_id: order.user_id,
            order_time: order.order_time
        })
    }
};

if (order.order_status === 'Confirmed') {
    decision = generateDecision("Prep");
} else if (order.order_status === 'PREPARED') {
    decision = generateDecision("Cook");
} else if (order.order_status === 'COOKED') {
    decision = generateDecision("Pack");
} else if (order.order_status === 'PACKED') {
    decision = generateDecision("Ship");
}

swf.respondDecisionTaskCompleted({
    taskToken: task_token,
    decisions: [decision]
}).promise()
    .then(console.log('Activity task scheduled'));
}

function handleOrder(userId, order_time, task_token) {
    let docClient = new AWS.DynamoDB.DocumentClient();
    let query_input = {
        TableName : "orders",
        KeyConditionExpression: "user_id = :id and
        order_time = :ts",
        ExpressionAttributeValues: {
```

```

        ":id":userId,
        ":ts":order_time
    }
};

docClient.query(query_input).promise()
    .then(data => orchestrate(data.Items[0], task_token));
}

swf.pollForDecisionTask(input).promise()
    .then(data => {
        if(data.workflowExecution) {
            let parts = data.workflowExecution.workflowId.
                split("_");
            let userId = parts[0]
            let order_time = parts[1]
            handleOrder(userId, order_time, data.taskToken);
        } else {
            console.log('Empty response when polling for
decision tasks')
        }
    });
});

```

One aspect to notice in this code is that I'm not passing any task list, which means that I have configured a default task list for the activity. The activity type version is the same for all the activities because I wanted to keep things simple for now. In the real world, activities and their versions diverge, so you need to make adjustments to use the correct version.

The entire orchestration logic happens inside the `orchestrate` method in the IF-Else block. Our code follows this happy path: `pollForDecisionTask` ➤ `handleOrder` ➤ `orchestrate` ➤ `generateDecision` ➤ `orchestrate` ➤ `respondDecisionTaskCompleted`. I have not handled any exception scenarios. This way, if there is an error in any of the methods, it will percolate to the main logic, where we can catch it and handle it at once.

However, this code doesn't repeat on a regular basis. This just runs once and handles one decision task and schedules an activity task and shuts down. If you want the decider to run and keep dishing out activities constantly, it has to be up and running all the time in an infinite loop. You can just follow the approach we used to create an SQS consumer.

We can surround this polling for decision tasks and orchestration logic in an infinite loop and start the program using process managers like **pm2** so that it keeps running in the background.

Run this code, and if you have a running workflow execution, this code schedules the activity work; and if you open the Activities tab of the workflow execution, you should see an activity being scheduled, and if that activity is not worked on soon enough, that will time out, and that shows up in the events list as well. If that happens, SWF will schedule a new decision task. It may look something like shown in Figure 4-9.

| Workflow Execution: eef94240-953c-11e9-8cdd-79d7b0b457b6_2019-06-24 02:42:29.146 | | |
|--|------------|--------------------------|
| Domain: RestaurantDomain | | |
| | Summary | Events |
| | Activities | Child Workflows |
| Event Date | ID | Event Type |
| Sun Jun 23 19:47:57 GMT-700 2019 | 7 | DecisionTaskScheduled |
| Sun Jun 23 19:47:57 GMT-700 2019 | 6 | ActivityTaskTimedOut |
| Sun Jun 23 19:42:57 GMT-700 2019 | 5 | ActivityTaskScheduled |
| Sun Jun 23 19:42:57 GMT-700 2019 | 4 | DecisionTaskCompleted |
| Sun Jun 23 19:42:56 GMT-700 2019 | 3 | DecisionTaskStarted |
| Sun Jun 23 19:42:29 GMT-700 2019 | 2 | DecisionTaskScheduled |
| Sun Jun 23 19:42:29 GMT-700 2019 | 1 | WorkflowExecutionStarted |

Figure 4-9. Events List After the Decider Worked Its Magic

How else can we build orchestration logic? By looking into the history of events. We can iterate over the entire event history to decide what activities are finished. In our workflow, by counting how many activity tasks are completed, we can understand where we are in the process, without fetching order state from DynamoDB. As our activity tasks are

Prep, Cook, Pack, and Ship, if no activities are completed, it means the order is just confirmed and yet to be prepared. The following snippet of code does just that:

```
function orchestrate(user_id, order_time, events, task_token) {  
    let completedActivities = 0;  
    for (let index in events)  
        if (events[index].eventType === "ActivityTaskCompleted")  
            completedActivities++;  
  
    let decision = {};  
    switch (completedActivities) {  
        case 0: decision = generateDecision("Prep"); break;  
        case 1: decision = generateDecision("Cook"); break;  
        case 2: decision = generateDecision("Pack"); break;  
        case 3: decision = generateDecision("Ship"); break;  
    }  
  
    swf.respondDecisionTaskCompleted({  
        taskToken: task_token,  
        decisions: [decision]  
    }).promise()  
        .then(console.log('Activity task scheduled'));  
}  
  
swf.pollForDecisionTask(input).promise()  
    .then(data => {  
        if(data.workflowExecution) {  
            let parts = data.workflowExecution.workflowId.  
                split("_");  
            let user_id = parts[0];  
            let order_time = parts[1];  
            orchestrate(user_id, order_time, data.events,  
            data.taskToken)
```

```
    } else {
        console.log('Empty response when polling for
decision tasks')
    }
});
```

In this code snippet, we are not making an extra call to DynamoDB, and the decision is made purely out of analyzing the history of events. However, we have to keep a few things in mind when going by this approach. Each pollForDecisionTask invocation just returns one page full of events, where the page size is determined by the maximumPageSize property.

If the workflow history has a lot of other events, it may not contain all tasks of the activity completed type; we have to make sure that we have all the events by looking at the nextPageToken property of the response. If it exists, that means we will have to make another call to SWF for getting the next page of history. You can also increase the page size to a big enough number that is most likely to be larger than most of the workflow execution sizes.

There is a slightly better version of the same implementation. We don't really have to look at all the events of type ActivityTaskCompleted. We can also make a conclusion based on the first event of the same type we encounter. This works because we got the events in reverse order, so that the first one we encounter in the events array is the last activity that occurred on the workflow. This logic in the following showcases just that:

```
let completedActivity = "";
for (let index in events) {
    let attributes = events[index].eventType.
                    activityTaskCompletedEventAttributes;
    if (attributes) {
        completedActivity = attributes.activityType.name;
```

```

        break;
    }
}

let decision = {};
switch (completedActivity) {
    case "" : decision = generateDecision("Prep"); break;
    case "Prep" : decision = generateDecision("Cook"); break;
    case "Cook" : decision = generateDecision("Pack"); break;
    case "Pack" : decision = generateDecision("Ship"); break;
}

```

Of course, even this has the issue of not getting even one event of the activity completed type in its first page; at least we don't have to get them all. Depending on the workflows you have in your project, the portion of executions that have too many events in history without even one activity task completed can be reduced by selecting a sufficiently large maximum page size.

Enough about orchestration logic, we so far haven't learned one aspect. What happens after a workflow is progressed from Prep to Cook to Pack to Ship and after the ship activity is completed? We should end the workflow execution. To end the workflow on a successful note, it is pretty similar to how we schedule an activity. Instead, we have to pass the decision of type CompleteWorkflowExecution to mark the closure of the workflow.

Implementing Activity Worker

Now that you have learned how to make decisions, it is time to start acting upon them. An activity worker is necessary for the real work to be performed for the function of the business. An activity worker is no different from an SQS consumer. It keeps polling for activity tasks that it can perform, and once it received an activity scheduled, it works on it and goes back to polling for the next activity.

Just like a decider, which needs to make calls to SWF through the method `pollForDecisionTask` constantly, an activity worker needs to invoke the `pollForActivityTask` method on the SWF client. This method requires us to pass an object of type `SWF.Types.PollForActivityTaskInput` as input. Here is the definition of the `PollForActivityTaskInput` interface:

```
export interface PollForActivityTaskInput {
    domain: DomainName;
    taskList: TaskList;
    identity?: Identity;
}
```

The property `identity` is to specify the activity worker's identification. It is an optional property but can help us debug issues such as which host is causing regular failures in case of debugging a production issue. The other two properties, `domain` and `taskList`, are mandatory. `TaskList` is to receive a particular set of activities rather than all the activities in a workflow.

An activity worker capable of performing packing of an item may not be able to ship it and vice versa. Depending on your application, a few workers of your application may actually be human workers who have to drive through the traffic to take the package to the customer who ordered it. So it is better to only poll for activities of a specific `taskList`. Go through the following code in Listing 4-6 to see a simple activity at work.

Listing 4-6. javascript-sources/chapter-4/demo-04/worker.js

```
const AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.SharedIniFileCredentials
    ({profile: 'admin'});

let swf = new AWS.SWF();
let ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
let docClient = new AWS.DynamoDB.DocumentClient();
```

```
async function prepareOrder (item) {
  let params = {
    TableName: "orders",
    Key: {
      user_id: { S: item.user_id },
      order_time: { S: item.order_time }
    },
    UpdateExpression: "set order_status = :val1",
    "ExpressionAttributeValues": {
      ":val1": {"S": "PREPARED"}
    },
  };
  console.log("Preparation Done, Saving the order");
  await ddb.updateItem(params).promise();
  console.log("Successfully prepared the data");

}

function handle(order, task_token) {
  console.log("Fetching the order.... ");

  let query_input = {
    TableName : "orders",
    KeyConditionExpression: "user_id = :id and
order_time = :ts",
    ExpressionAttributeValues: {
      ":id":order.user_id,
      ":ts":order.order_time
    }
  };
  docClient.query(query_input).promise()
    .then(data => prepareOrder(data.Items[0], task_token));
  let params = {
```

```
    taskToken: task_token,
    result: 'PREPARED'
};

swf.respondActivityTaskCompleted(params).promise()
  .then(data => console.log("Successfully completed the
    task: " + JSON.stringify(data)));
}

let input = {
  domain: "RestaurantDomain",
  taskList: {
    name: "PREP_LIST"
  },
};

swf.pollForActivityTask(input).promise().then(data => {
  handle(JSON.parse(data.input), data.taskToken);
});
```

The execution begins in the bottom three lines of code with `pollForActivityTask`. Once the promise returns some data, the responsibility moves to the `handle` method where the order is fetched. Once the order is fetched, it is processed; in other words, its status is set to PREPARED. Once the status is set to PREPARED, the worker sends a response of type `ActivityTaskCompleted`. The input for this type of response is `taskToken` and `result`. `Result` is just another string to pass some additional information.

When our activity worker is trying to update the order, what if the order status is already cancelled? It can happen if the customer cancels the order. There is no point in trying to complete all the work on cancelled orders. So we can set a check before we update the order status to PREPARED and skip updating it if the order is already cancelled. In such cases, we can set the `result` to CANCELLED or something to that effect.

This information can be used by the decider to cancel the workflow execution or any child workflows we triggered through the parent workflow. This way, we can handle the cancelling of the orders without ever needing to fetch the order status in a decider. There is nothing wrong with fetching the order status from database in decider; sometimes, it makes sense to do that.

What if no activities are ever performed on a specific order? The result will never be CANCELLED, even though in the database the order is set to CANCELLED before the first decider starts its operation on that workflow. The customers can be as fickle as they want and cancel immediately. In such cases, there is no point in scheduling an activity. Or if there is some issue with workers and they are offline for a very long time and the orders are cancelled from the back end, the deciders will keep scheduling activities that get timed out. In such cases, the decider can be benefitted from fetching the order state.

So there is a case to be made for either use case. You may take a median approach, wherein if the history of events has a lot of timeouts or failures, you can fetch the order state from DynamoDB and decide accordingly. You can pick any or every approach to solve the problems you encounter.

Serverless Workflow

So far, we have built the workflows that run entirely on servers. The decider and activity workers need to poll for decision and activity tasks, respectively, constantly. This may look as if we need servers for it. Since 2015, SWF started allowing lambda activity workers, which means some of the work that needs to be performed in a workflow can be changed to lambda function.

Even deciders can be made serverless by following a few hacks like using CloudWatch events to trigger a lambda function on a regular basis. In this section, we will see how to build serverless deciders and activity workers.

Serverless Activity Workers

In the previous section, we have seen how to build an activity worker. It is very nice, but for the activities to be worked on, an activity worker needs to keep running in the background. This prevents us from going serverless and increases our costs. To run the majority of our application without maintaining our servers, our deciders can schedule a lambda activity instead of a regular activity.

If you go back to Table 4-1, one of the valid decision types is ScheduleLambdaFunction. By using this decision type, we can provide ARN or name of the lambda function to be triggered. If we are providing name, we have to make sure that the lambda function is in the same AWS account and region. The following code snippet schedules a lambda activity:

```
function generateDecision(activityName) {
    return {
        decisionType: "ScheduleLambdaFunction",
        scheduleLambdaFunctionDecisionAttributes: {
            id: " " + Math.random(),
            name: 'lambda-activity-' + activityName,
            input: JSON.stringify({
                user_id: user_id,
                order_time: order_time
            })
        }
    };
}
```

When we create a decision of type ScheduleLambdaFunction, we need to pass the map of type scheduleLambdaFunctionDecisionAttributes. This map needs id and name parameters. The parameter id is just a string that identifies a lambda function execution in the event history. The name parameter can take an ARN or name of the lambda function.

In the preceding code snippet, we just created a random number to be the ID. Instead of this, we may be able to provide a counter or timestamp or such to make better use of this parameter. The name we have provided turns into lambda-activity-Prep, lambda-activity-Cook, and so on. The expectation is that there are already lambda functions of these names in our AWS account.

There are three more parameters that this decision accepts. They are input, control, and startToCloseTimeout. These are optional parameters. The parameter input is to pass input to the lambda function, which can take a string. So we need to stringify if we want to pass a complex object.

The other parameter control also is an input, but this is not passed to the lambda function. Instead, it can be read by the next decider when it is going through history. It may be some information like how many times a specific lambda activity is retried or such. This may be used to terminate the workflow if we decide that retrying the same activity five times is critical failure. The final parameter startToCloseTimeout is just a number of seconds after which this lambda activity must be considered as timed out if not completed.

If you run this decider, the activity may be scheduled, but the lambda function may not get executed, and you may end up with an event history as shown in Figure 4-10.

| Event Date | ID | Event Type |
|----------------------------------|----|-----------------------------|
| Wed Jun 26 22:56:55 GMT-700 2019 | 8 | WorkflowExecutionTerminated |
| Wed Jun 26 21:00:32 GMT-700 2019 | 7 | DecisionTaskScheduled |
| Wed Jun 26 21:00:32 GMT-700 2019 | 6 | StartLambdaFunctionFailed |
| Wed Jun 26 21:00:32 GMT-700 2019 | 5 | LambdaFunctionScheduled |
| Wed Jun 26 21:00:32 GMT-700 2019 | 4 | DecisionTaskCompleted |

| | |
|---|---|
| StartLambdaFunctionFailed [with EventId 6 selected] | |
| Cause | ASSUME_ROLE_FAILED |
| Event Timestamp | Wednesday, June 26, 2019 9:00:32 PM UTC-7 |

Figure 4-10. Lambda Trigger Failure

Click the StartLambdaFunctionFailed event. The details of the event give a reason as to why SWF was unable to start the lambda function. It is because the SWF couldn't assume the role that can trigger a lambda function. If you remember the creation of a workflow type, we have not provided any lambda function role in the second page of the workflow type creation wizard, as shown in Figure 4-11.

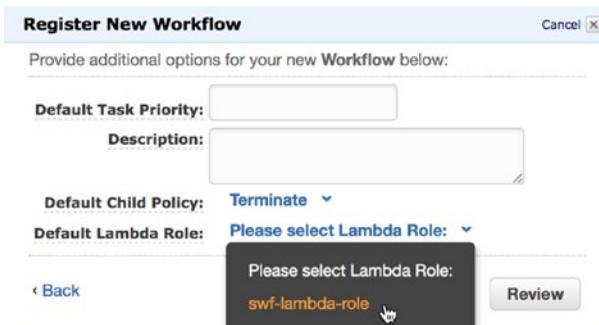


Figure 4-11. Selecting Default Lambda Role

If you have just followed the previous step, you wouldn't find any role in the Default Lambda Role dropdown. We need to create a role that can be assumed by the SWF service. The role should have access to trigger lambda functions. Creating this role is slightly different, but not so difficult.

Open the IAM service and select the Roles hyperlink in the left-side navigation bar. Once the Roles page is shown, click the Create role button to open a new wizard. Both the Roles page and create role wizard are shown in Figure 4-12. The wizard allows you to create a role with a trusted entity. A trusted entity is a service that can assume this role.

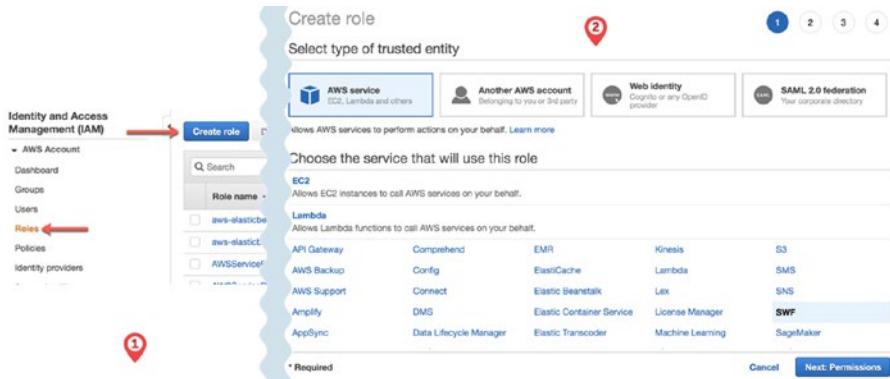


Figure 4-12. Creating Role with SWF as Trusted Entity

Select the AWS service and click SWF in the list of services. Once selected, click the button Next: Permissions. This navigates to a page where we can attach IAM policies. AWS Lambda Role may already have been selected, select it otherwise. Move on to the next pages until you get to a page where you can provide a name for the role. Provide the name `swf-lambda-role`. Click the Create role button to finish the process of role creation.

Once you do this, go back to the workflow type register wizard. You can fill in all the details on the first page and go to the next page and click the Default Lambda Role dropdown. You should see the new role you have created. Select the role to give the SWF service access to the Lambda service upon this workflow execution.

However, you cannot modify a workflow once you create it. You can only create a new workflow with the same name but with a different version. You have to make sure that you change the lambda function, starting the order workflow to use the new version number. Now, when the decider schedules a Lambda activity, it will be executed without any issues.

Serverless Deciders

A decider needs to run continuously and keep scheduling activities. An activity can be easily converted to a lambda function because a decider can schedule it. However, there is no one scheduling a decider to start working. We are just running a decider program with orchestration logic in a loop using a process manager like pm2.

To convert it to a lambda function, we need to find out a way to invoke the lambda function continuously. Do we need another program to do just that? Luckily, we don't have to. AWS has something called CloudWatch events, using which we can trigger lambda functions. We can use CRON-styled scheduling in CloudWatch events to invoke the lambda function. Let us see how to do that. But before we can do that, let us see how to convert our decider program into a lambda function. Look at the following code in Listing 4-7 to see what is to be done.

Listing 4-7. javascript-sources/chapter-4/demo-05/lambda-decider.js

```
const AWS = require("aws-sdk");

exports.handler = async (event, context) => {
    let swf = new AWS.SWF();

    let input = {
        domain: "RestaurantDomain",
        taskList: {
            name: "OrderTasks"
        },
        maximumPageSize: 250,
        reverseOrder: true
    };
}
```

```

function orchestrate(user_id, order_time, events,
task_token) {...}

await swf.pollForDecisionTask(input).promise()
    .then(...Call Orchestrator...);

};


```

The code is pretty much the same except for the fact that we introduced exports.handler just like any other lambda functions we have seen earlier. I truncated the code in orchestrate and promise handler. You can just copy the code from the previous decider.

Another change is that we added an await keyword in front of the pollForDecisionTask line. This ensures that Lambda doesn't terminate before polling is finished. As you know, polling for a decision task is a long polling and takes 60 seconds to complete in the case of no decision tasks; it is necessary that you give the Lambda function timeout of at least 60 seconds as shown in Figure 4-13. Slightly more is better because if a decision is received after the 59th second and the rest of the orchestration and responding takes a few more seconds, the Lambda function would time out just before the work is complete.

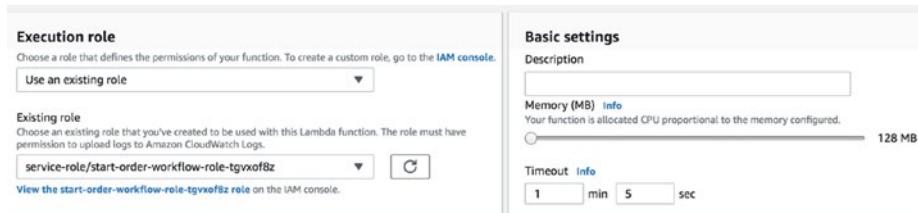


Figure 4-13. Lambda Role and Timeout Configuration

When you create and save this lambda function, you need to make sure that you give a role that has access to SWF functionalities; otherwise, executing this lambda function gives an access denied error. You should be very familiar with all these access-related aspects of AWS by now. If not, refer to the first chapter on how to do these.

Now, let us look at the part where we see how to trigger this lambda function on a regular basis. As I mentioned earlier, we use CloudWatch events for that. Go to the Designer tile of the Lambda function to access the Triggers section, as shown in Figure 4-14. If you click CloudWatch Events, it gets added to the list of triggers.

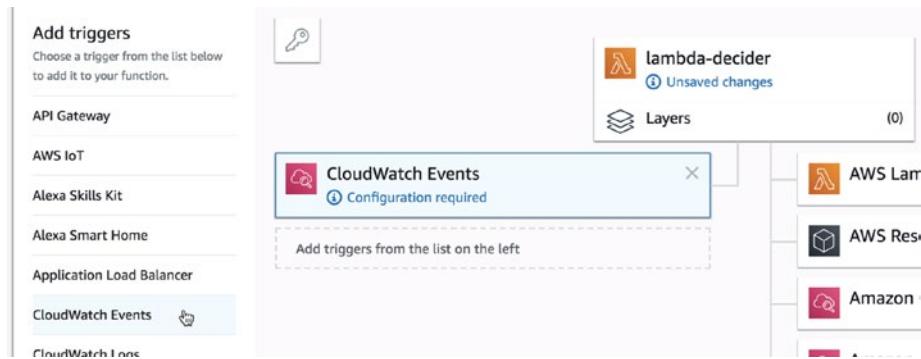


Figure 4-14. Click CloudWatch Events to Add a Trigger

CloudWatch events require a certain configuration to be made. Click the CloudWatch Events tile to expose the necessary configuration. You may have to scroll down to see the configuration options, as shown in Figure 4-15. You may not have any rules so far. In any case, select Create a new rule from the first dropdown to start creating a new rule.

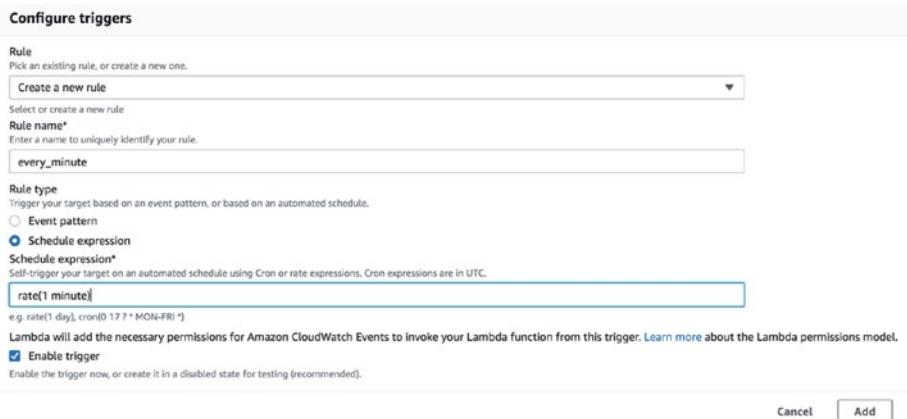


Figure 4-15. Rate Expression

Provide a name meaningful enough to depict a rule that triggers every minute. I gave every_minute. Now, you can select either Event pattern or Schedule expression. We need to write a schedule expression because we want to trigger this based on a schedule. Event pattern can be selected if we want to use events such as EC2 auto-scaling, S3, and so on.

We can have two types of schedule expressions: a rate expression and a cron expression. Cron expression takes the following format. If you are familiar with cron expressions, the format should be quite familiar to you. I don't want to go into the details of how to use cron expressions as we will use rate expressions in this example:

cron(Minutes Hours Day-of-month Month Day-of-week Year)

Rate expressions can be used if we want a simple way to specify a rate instead of a complex cron expression. A rate expression has two mandatory fields: the value and the unit that should be applied to the value. So if you want to specify a schedule of every 2 hours, you just have to write rate(2 hours) in the expression text box:

rate(value unit)

As we wanted to execute this lambda function every minute, we just have to write it as rate(1 minute). Make sure that the Enable trigger button is checked as shown in the preceding figure. Click the Add button to complete schedule configuration. Save the Lambda function once the schedule is created. This will immediately start running the lambda function every 1 minute.

However, as you remember, we are running the lambda function for the whole 1 minute, so if we run the lambda function every 1 minute, even before the previous lambda function finished its execution, a new execution will start. This means that our decider will be running all the time. You may want to adjust the rate based on your need.

If you wish to let the lambda function sleep for a minute, you can give the rate as 2 minutes. That way there is a gap of 1 minute between each invocation. However, keep in mind that the lambda function will exit as soon as it received one decision task. So if we receive an event in the first second and the lambda function exits immediately, based on the rate expression, the lambda will be sleeping for a whole 2 minutes. Keep these parameters in mind before you decide on a rate.

If you want to know whether the lambda function is being triggered at the given rate or not, you can go to the Monitoring tab of the lambda function after some time. You should see a few graphs as shown in Figure 4-16. The first graph is the number of invocations for each period of 5 minutes. As we have set the rate to be 1 minute, there would be 5 invocations for each period.

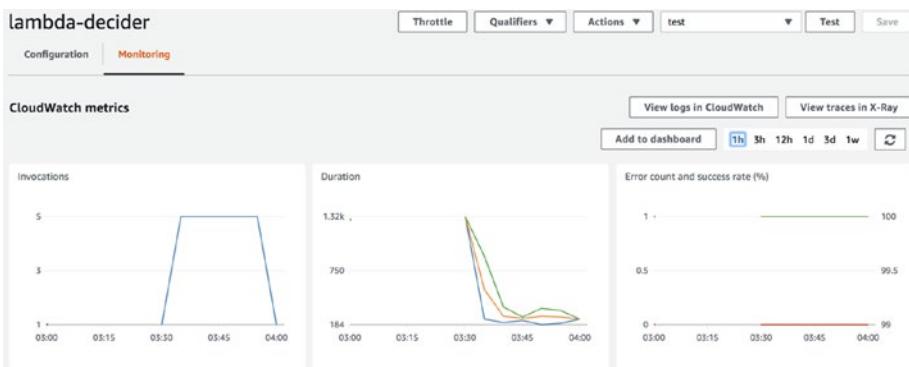


Figure 4-16. Invocations on Regular Basis

The next graph is the total duration in seconds. As each invocation is 60 seconds, the total duration will be the sum of all invocations in that period. You may see more than 300 because I may have run it multiple times and the way lambda measures time period is slightly different. The final graph shows the errors and success of lambda invocations.

I suggest you disable the CloudWatch trigger as soon as you complete this exercise. Lambda alone may not cross the threshold of the free tier, but as you are using other services as well, you may accidentally forget about this and keep burning a hole in your pocket when you also use other services.



Caution Make sure that you disable the trigger. Otherwise, the lambda function will be running for a whole month, and you may cross over the free tier limits.

With this, we have completed creating a serverless decider that purely runs On Lambda. However, keep in mind that constantly running lambda is costlier than having a dedicated server if you calculate correctly. There are a lot of variables such as allocated memory and timeout that influence how much you will be charged. Creating a serverless activity worker is beneficial because you don't have to keep it running.

Advanced Topics

Timers

There are times when you need to set timers in a workflow to make sure that you make decisions based on the results of the timer. For example, in our restaurant web site, if we don't deliver to a customer within 1 hour, we may want to refund 10% of the total amount. We can start a timer as soon as we trigger prep activity, and if the timer expires before ship activity is completed, we can schedule a new activity for refund.

To use timers, we need to first send a start timer decision with timeout in seconds. SWF keeps track of the number of seconds elapsed since the decider has sent the decision. After a given number of seconds, SWF records an event of type TimerFired. When a decider gets the event history, it also gets a timer fired event in the event history.

This can be achieved by scheduling two decisions instead of one when scheduling a prep activity. The first decision schedules the prep activity, and the second decision starts a timer. Both will be executed in parallel. Let us see how to do that now.

Start Timer

Starting a timer is no different from scheduling a new activity. Very minor changes are required. We just have to change the decision type and provide the required attributes. The following is an example decision that can be created to respond with a StartTimer decision:

```
{  
    decisionType: "StartTimer",  
    startTimerDecisionAttributes: {  
        timerId: " + Math.random(),  
        startToFireTimeout: 3600  
    }  
}
```

This decision starts a timer for 1 hour. As respondDecisionTaskCompleted takes an array of decisions, we can just add the prep decision and timer decision into one array and pass them along to SWF. We need to pass start timer decision attributes for a decision of type StartTimer:

```
function startTimerDecision() {  
    return {  
        decisionType: "StartTimer",  
        startTimerDecisionAttributes: {  
            timerId: " + Math.random(),  
            control: JSON.stringify({  
                user_id: user_id,  
                order_time: order_time  
            }),  
            startToFireTimeout: "3600"  
        }  
    }  
}  
  
let decisions = [];  
switch (completedActivity) {  
    case "" : decisions = [generateDecision("Prep"),  
    startTimerDecision()]; break;  
    case "Prep" : decisions = [generateDecision("Cook")];  
    break;  
    case "Cook" : decisions = [generateDecision("Pack")];  
    break;  
    case "Pack" : decisions = [generateDecision("Ship")];  
    break;  
}  
}
```

CHAPTER 4 SIMPLE WORKFLOW SERVICE

```
swf.respondDecisionTaskCompleted({
    taskToken: task_token,
    decisions: decisions
}).promise()
.then(console.log('Activity task scheduled'));
```

This code snippet constructs the decisions array in the orchestration logic. The startTimerDecision is called upon when the order is about to be prepped. The attributes for this type of decision include mandatory timerId and startToFireTimeout. Both the parameters are strings.

The parameter timerId can be used to identify when the timer fires. The TimerFired event has the same id passed in along with the start timer event. A workflow may have more than one timer, so it is essential for us to recognize which timer has elapsed. Hence, the timer id can be used for that purpose. It is depicted in Figure 4-17. The timer IDs of both TimerStarted and TimerFired events are the same as you can see in the screenshot in the following.

| Event Date | ID | Event Type |
|--|----|--|
| Sat Jun 29 12:07:44 GMT-700 2019 | 10 | TimerFired |
| Sat Jun 29 12:07:14 GMT-700 2019 | 9 | DecisionTaskScheduled |
| Sat Jun 29 12:07:14 GMT-700 2019 | 8 | LambdaFunctionFailed |
| Sat Jun 29 12:07:14 GMT-700 2019 | 7 | LambdaFunctionStarted |
| Sat Jun 29 12:07:14 GMT-700 2019 | 6 | TimerStarted |
| Sat Jun 29 12:07:14 GMT-700 2019 | 5 | LambdaFunctionScheduled |
| Sat Jun 29 12:07:14 GMT-700 2019 | 4 | DecisionTaskCompleted |
| Sat Jun 29 12:07:14 GMT-700 2019 | 3 | DecisionTaskStarted |
| Sat Jun 29 12:07:10 GMT-700 2019 | 2 | DecisionTaskScheduled |
| # | | |
| TimerStarted [with EventId 6] selected | | |
| Control | | [{"user_id": "01e715f0-69cb-11e9-b61b-3146d2119:07:10:177"}] |
| Decision Task Completed Event Id | 4 | |
| Event Timestamp | | Saturday, June 29, 2019 12:07:14 PM UTC-7 |
| Start To Fire Timeout | | 30 seconds |
| Timer Id | | 0.7412146475421022 |

| Event Date | ID | Event Type |
|---------------------------------------|----|---|
| Sat Jun 29 12:07:44 GMT-700 2019 | 10 | Time-Fired |
| Sat Jun 29 12:07:14 GMT-700 2019 | 9 | DecisionTaskScheduled |
| Sat Jun 29 12:07:14 GMT-700 2019 | 8 | LambdaFunctionFailed |
| Sat Jun 29 12:07:14 GMT-700 2019 | 7 | LambdaFunctionStarted |
| Sat Jun 29 12:07:14 GMT-700 2019 | 6 | Time-Started |
| Sat Jun 29 12:07:14 GMT-700 2019 | 5 | LambdaFunctionScheduled |
| Sat Jun 29 12:07:14 GMT-700 2019 | 4 | DecisionTaskCompleted |
| Sat Jun 29 12:07:14 GMT-700 2019 | 3 | DecisionTaskStarted |
| Sat Jun 29 12:07:10 GMT-700 2019 | 2 | DecisionTaskScheduled |
| # | | |
| TimerFired [with EventId 10] selected | | |
| Event Timestamp | | Saturday, June 29, 2019 12:07:44 PM UTC-7 |
| Started Event Id | 6 | |
| Timer Id | | 0.7412146475421022 |

Figure 4-17. Start Timer vs. Timer Fired

Along with these parameters, we can also pass the control attribute to pass along additional information to the workflow execution. This may sometimes be necessary to make more decisions. In our use case, we don't

need it. Because of the way we created the workflow execution id, we are getting the same information. If you want to send any more information, you can use the control attribute.

Cancel Timer

In our application, we are starting the timer when scheduling the first activity. What if our order workflow is more complex? What if we have one more step in our workflow that is targeted to gather feedback, but our timer has no use once it has been shipped (Figure 4-18)?

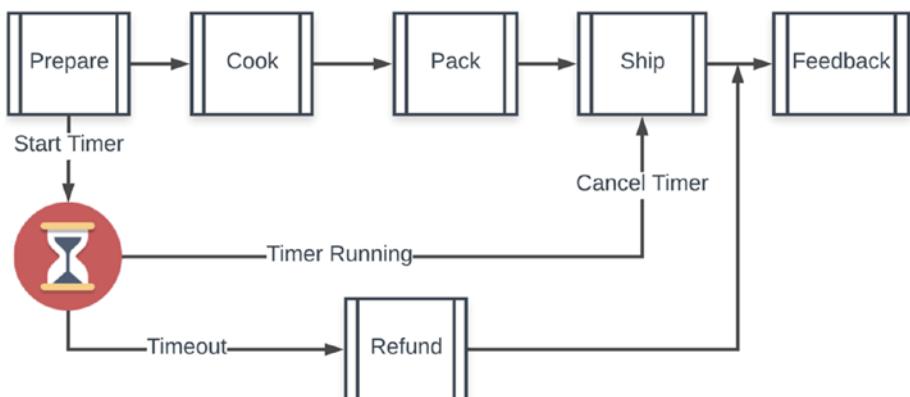


Figure 4-18. Workflow with Timer

We need to cancel the timer if the shipping is done in less than an hour, but the workflow can only be closed if the feedback task is finished. In this task, we call the customer to find if there are any issues with the order. It is not really necessary to cancel the timer when the TimerFired event is triggered. We can check the order status, and if the order is already shipped, we ignore it; otherwise, we schedule a refund activity.

CHAPTER 4 SIMPLE WORKFLOW SERVICE

We don't always have the luxury of waiting for the workflow to finish or ignore the TimerFired event. The workflows may be more complex, and we need to cancel the timers without failing. To do that, we can send the decision of type CancelTimer. Here is the example of a cancel timer decision:

```
let timerId = {};
for (let index in events) {
    let attributes = events[index].timerStartedEventAttributes;
    if (attributes) {
        timerId = attributes.timerId;
        break;
    }
}

function cancelTimerDecision(timerId) {
    return {
        decisionType: "CancelTimer",
        cancelTimerDecisionAttributes: {
            timerId: timerId
        }
    }
}

let completedActivities = 0;
for (let index in events) {
    let attributes = events[index].eventType.
        activityTaskCompletedEventAttributes || 
    events[index].eventType.lambdaFunctionCompletedEvent
    Attributes;
    if (attributes) {
        completedActivities++;
    }
}
```

```

let decisions = [];
switch (completedActivity) {
    case 0 : decisions = [generateDecision("Prep"),
        startTimerDecision()]; break;
    case 1 : decisions = [generateDecision("Cook")]; break;
    case 2 : decisions = [generateDecision("Pack")]; break;
    case 3 : decisions = [generateDecision("Ship")]; break;
    case 4 : decisions = [cancelTimerDecision(timerId),
        generateDecision("Feedback")]; break;
    case 5 : decisions = [workflowCompletedDecision()]; break;
}

```

All this decision needs is the timer id. We can find the timerId by iterating over all the event history until we get to the timer started event. The preceding code snippets show how to do that. Once we get the timer id, we generated cancel timer and passed the timer id to the attributes object.

Recording Heartbeats

Failures happen and systems may run slow. Irrespective of how much capacity we may provide, it is possible that tasks take more time than usually expected. Even when the tasks are successful, they may take longer than normal. Sometimes it is necessary for us to fail fast to ensure that our business runs as usual. Otherwise, we may end up waiting too long when we can be doing critical operations. In businesses like online ordering of food, the operations are time sensitive.

To solve this problem, SWF introduced a concept of heartbeats. Heartbeats are a mechanism for activities to let SWF know that the task is progressing. Any activities that take longer than a few seconds need

to send signals to SWF regularly. This signaling is nothing but recording heartbeat. An activity worker can record heartbeat using the method recordActivityTaskHeartbeat:

```
swf.recordActivityTaskHeartbeat({  
    taskToken:task_token,  
    details: "Delivery Executive Picked Up The Order"  
});
```

The preceding snippet shows how to send regular signals to SWF. When scheduling an activity, we can specify a heartbeat timeout. When the activity is being worked upon, the worker has to record heartbeat on a regular basis. If SWF doesn't receive heartbeats for more than the specified number of seconds as per the activity task schedule, that task is considered as timed out.

Heartbeat timeout may be configured when creating an activity type or when scheduling an activity. As always, if the timeout is provided in both the cases, the one at the time of scheduling an activity takes precedence. Maybe a few days in a year are peak days, and your systems are expected to run slow. In that case, you want to give allowance to longer timeout. This can be done easily by specifying the timeout when scheduling the activity as shown in the following code snippet:

```
{  
    decisionType: "ScheduleActivityTask",  
    scheduleActivityTaskDecisionAttributes: {  
        activityId: activityName + "_" + Math.random(),  
        activityType: {  
            name: activityName,  
            version: "0.3"  
        },  
    },
```

```

input: JSON.stringify({
    user_id: order.user_id,
    order_time: order.order_time
}),
heartbeatTimeout: 300
}
}

```

As per the preceding snippet, SWF considers that activity has timed out if the worker doesn't respond within 300 seconds.

Miscellaneous Topics

All these sections demonstrate that you can build complex workflows without maintaining tons of queues and making use of many other services. However, SWF is not without its quirks. One of the biggest issues with SWF is that the user has to manage the infrastructure as it is not truly serverless. You are supposed to maintain your own servers that run this orchestration logic.

Another issue is that the orchestration logic is not easy to maintain as it may distribute across several deciders and there is no single window through which we can get a grasp of what is going on. Each decider may poll for decision tasks on a separate task list making it difficult for us to see the big picture.

Step Functions solves many of these challenges accompanied with using the Simple Workflow Service. In almost all cases, Step Functions outshines SWF by miles. We will look at advantages and how easy it is to use Step Functions in the next chapter. However, before the public release of Step Functions, many people have faced the challenges and solved them by creating a few frameworks. These frameworks simplify the task of working with SWF.

Frameworks

As you know, developing SWF applications is a bit involved process. It can get overwhelming. We can certainly develop the applications using the vanilla approach. However, many people who start developing SWF applications end up creating some other frameworks to simplify some of the development and deployment activities.

AWS FLOW FRAMEWORK

This is a framework developed by Amazon and makes use of AspectJ load time and compile time weaving to auto-generate some client classes. Using this framework, we can concentrate on writing core logic using annotations.

However, this framework is too old, and there are not many examples that work with latest versions of Java, and even the example provided in their GitHub repository expects that you set up the project in Eclipse IDE. But I don't see much benefit out of following this framework. Irrespective of what the framework documentation claims, there is a lot of boilerplate code.

Setup is quite difficult to follow. You can try your luck here:

- <https://github.com/aws/aws-swf-flow-library>
- <https://docs.aws.amazon.com/amazonswf/latest/awsflowguide/welcome.html>

Netflix Glisten

Glisten is a wrapper around the Flow Framework and allows us to code and takes help from Groovy to hide the boilerplate code. It is developed by Netflix and eventually made into an open source framework for the benefits of developers trying to use it to build SWF applications. You can find the information about this framework here.

- <https://netflix.github.io>

WaterFlow

As per the web site of this framework, this is a non-magical, easy to understand JDK8 framework for use with the Simple Workflow Service as per the framework's home page. Unfortunately, the author seems to have abandoned the project, and I can no longer find the maven repository artifacts.

As you can see, there are very few frameworks available, and some of them are either abandoned or not getting many updates anymore. The frameworks that still work are not well documented to work with latest technologies like JDK8. For all intents and purposes, you are better off working with the vanilla approach if you want to use SWF for any reason. However, make sure that you try Step Functions before you attempt to use SWF.

Pricing

SWF has a free tier under which we have 1000 workflow executions free of charge and 10000 tasks, timers, and signals for a whole month. Thirty thousand workflow days can be used free of charge. A workflow day is defined as a 24-hour period of time during which a workflow execution is running. If 20 workflow executions were open for 12 hours each, they would spend $(20 * 0.5 \text{ days})$ 10 workflow days. So you can have 1000 workflow executions that can take up to 24 hours each without any cost. Remember that the limit of 10000 tasks, timers, and signals may be crossed even if you don't cross workflow execution days.

Once you have spent the free tier, in the us-west-2 region, you are charged \$0.0001 per execution when the workflow starts. For continued executions and retention of the workflow, you are charged \$0.000005 for every workflow day. For each task or signal or timer, it costs \$0.000025. In addition to this, you are also charged for the data transfer out of the service. Each call or poll or signal will consume some data that is charged \$0.09 per GB; your mileage may vary based on how much data you consume.

However, this is not where the costs end. You may be running activity workers and deciders on servers that are always up and running. Those servers also cost money. If you are restricting yourself for the examples in the book, you shouldn't go beyond free tier even for those. Just make sure you shut down the instances as soon as you are done working on examples.

Assume that we have 20000 orders on our web site for each month. Each order workflow takes 4 hours to finish, including the refund timer. There are four activities, one timer on an average for each workflow execution. That is five 100000 tasks. We poll for decisions every 2 minutes, and each call gets 10 KB of data.

We are assuming that we retain workflows for seven days after execution. Assume that we are running one t2.small 1-year reserved instance for each activity and one m5.large for running a decider. The total cost will be less than \$100 for each month for these assumptions as shown below. This doesn't include the costs of DynamoDB and data storage. Majority of the cost is for the servers. If we go serverless, we can reduce it even further, especially for activity workers.

- Workflow initiation cost – $0.0001 * 20000 = \$2$
- Workflow execution cost – $0.000005 * 20000 * 4/24 = \0.0167
- Workflow retention cost – $0.000005 * 7 * 20000 = \$0.7$
- Task and signal cost – $0.000025 * 100000 = \$2.5$
- Data transfer cost – $0.09 * 21600 * 10 \text{ KB} = \0.21
- EC2 instance costs for workers – $0.013 * 24 * 30 = \$37.44$
- EC2 instance costs for decider – $0.057 * 24 * 30 = \$41.04$

As always, these costs are per region. As I mentioned earlier, the prices keep changing. Most of the times, the trend is downward based on history of AWS prices. So make sure you reconfirm the pricing based on your region by going to this pricing page of the SWF and other services you are planning to use.

Limits

By default, AWS sets some limits on the usage for your own good. You can always request AWS to raise the limits if you really need it. These limits are per account. There are many limits. Listing all of them here is not going to be a very effective usage of the page, so some of the limits are as follows:

GENERAL LIMITS

1. Maximum domains – 100.
2. Maximum workflow and activity types – 10000 per domain (both registered and deprecated).
3. Maximum open workflow executions – 100000 per domain.
4. Maximum workflow execution time – 1 year.
5. Workflow retention limit – 90 days.
6. Maximum pollers – 1000 per task list.
7. Maximum open activities – 1000 per workflow execution.
8. Maximum open timers – 1000 per workflow execution.
9. Maximum decisions in a decision task response – Count may vary, but total data size can be 1 MB.

Throttling Limits

Throttling limits specify how many individual calls can be made. They accrue in a bucket at a fixed refill rate. The bucket is for burst loads. Once we consume all the available units in the bucket, we need to wait for more units to be refilled before we can use the API:

1. Poll for decision tasks – Refill @ 12/sec. Max 1000
 2. Poll for activities – Refill @ 10/s. Max 1000
 3. Schedule activity task – Refill @ 10/s. Max 100
 4. Start timer – Refill @ 25/s. Max 500
 5. Register domain – Refill @ 1/s. Max 50
-

Summary

In this chapter, we have understood how to build workflows using the Simple Workflow Service of AWS. We have seen how we can keep the whole execution details together, unlike doing it with the Simple Queue Service. With queues, we get the same orchestration results provided that we do enough coding, but the event history is not easy to summarize.

If there is an issue with an order of a customer, wading through each activity log is going to be a tough task. With SWF, it is easy to see what is going on with one particular order because all of them are tied together as one entity. We can fetch them with one call or can use the AWS console to observe the execution history.

We have begun this chapter by setting up types necessary for a workflow to function. We started with registering a domain, workflow type, and activity type. The types are just names that are registered in the SWF service; their implementations come next. We have implemented the actors of an SWF workflow in the next section.

We have started with implementing a starter and seen how to start a workflow execution. Once the execution of a workflow starts, SWF schedules a decision task. The workflow will be waiting for a decider to poll for the decision task and decide on what happens with the workflow next.

We have then implemented a decider that pulls the workflow event history and analyzes the state of the order and decides on which activity has to be scheduled next. We have also seen how to take different approaches to orchestration logic. The other approach we can take is by analyzing the event history alone. We have seen that as well in the next demo.

Deciders are meant for scheduling activities that perform the business functions. Activity workers keep running on a server and continuously poll for activity tasks scheduled by deciders. We have seen how to develop activity workers using JavaScript and NodeJS. The activity workers do their job and respond to SWF with the result. Any failure in performing activities results in them either timing out or getting rescheduled by the next decider's task.

We have also seen how to schedule lambda function invocations and necessary configurations to allow SWF to invoke lambda functions. Once we have all the components in the SWF framework built, we have seen how to mark the workflow as completed once all the activities are completed.

Once we have seen how to build the components and run them on servers, we have also seen how to build totally serverless actors with the help of Lambdas and CloudWatch events to regularly schedule the decider lambda function. Even though we can run deciders as well on Lambda function, it is not going to be very economical.

In the final part of the chapter, we have seen some miscellaneous topics such as timers and recording heartbeat from the activity workers. We have also learned about a few frameworks available that are intended to simplify workflow development. We have also seen information on the pricing, throttling, and limits set by AWS.

CHAPTER 4 SIMPLE WORKFLOW SERVICE

In the next chapter, we will see how to develop similar workflows using Step Functions, which is like the Simple Workflow Service on steroids. With all its bells and whistles, SWF does have its limitations, and Step Functions solves them better.

CHAPTER 5

Step Functions

A state machine is any device storing the status of something at a given time. The status changes based on inputs, providing the resulting output for the implemented changes. A finite state machine has finite internal memory. Input symbols are read in a sequence producing an output feature in the form of a user interface.

—Techopedia Staff

In all the previous chapters, you have learned different ways to orchestrate components in your application for order processing, starting with simple queue-based orchestration to building simple workflows to keeping the workflow of an entire order together. If we are creatures of simple desires and needs, that is where we would stop. However, the brilliant minds at Amazon are not the ones to settle.

They have started jotting down the issues arousing out of using the Simple Workflow Service at a large scale and are successful in seeing issues with the service. Number one pain point with SWF is that there is no modeling language. Building orchestration components is a fairly disjointed process. We need to look at all the deciders to understand the flow.

The other issue is that there is no visualization of workflow execution. To understand where we are in a workflow, we need to look at the event history to get a grasp of where we are. Just relying on textual information

in the AWS console is neither simple nor intuitive. Another issue is that the deciders have to constantly run polling for new decision tasks leading to higher EC2 costs. Step Functions solves these issues fairly neatly.

Key Concepts

Step Functions Service (SFn) enables you to coordinate components of your application using visual workflows. SFn provides a reliable way to coordinate components without problems of SWF weighing down our applications, creating bottlenecks. With SFn, you diagnose and debug production issues with ease. To understand the usage of this service, we need to understand some important terminology.

State Machine

Step Functions is based on concepts of the state machine. However, what is a state machine? Simply put, a state machine is just a collection of states, the relationship among the states and their inputs and outputs. A state machine is an abstract device that can be in any one of the predefined states at any of the given point in time. Take the example of any order in our order processing system. An order can be in one of the following states at any point in time:

- CONFIRMED
- PREPARED
- COOKED
- PACKED
- SHIPPED
- COMPLETED

An order that is being prepped cannot be in a state of being shipped. The transition is also fixed. It can never go from COOKED to PREPARED or even from COOKED to SHIPPED directly. It has to transition from COOKED to PACKED to SHIPPED state. So our order processing system is nothing but a state machine.

Tasks

Work done in a state machine to transition the system from one state to another is a task. In our order processing state machine, a delivery executive needs to physically perform the task of taking the order packet from the restaurant to the customer's home.

Tasks are how states get the work done in any system. Step Functions allows you to coordinate individual tasks by expressing your workflow as a finite state machine, written in the Amazon States Language. Tasks perform work with the help of an activity or a Lambda function or by invoking an API of another service and passing parameters.

Activity

An activity can be an application that waits for an operator to perform some action or a component deployed on EC2 or ECS that scans an S3 bucket for a new file to be uploaded or listens to an SQS queue or SNS topic for a message to be published.

Just like in SWF, activities poll Step Functions regularly with the help of GetActivityTask API to receive new activity tasks and send task status using SendTaskSuccess, SendTaskFailure, and SendTaskHeartbeat APIs – very similar to how SWF operates.

Basic Operations

Creating State Machine

As with every other service, AWS allows you to create a state machine with ease. We can create a sample state machine in a few simple clicks from the SFn console. It can be done in two simple steps. In step 1, we need to specify a definition and name, and in step 2, we need to provide configuration information such as IAM role required by the step functions to execute the state machine.

Irrespective of the steps mentioned, there is a step 0 that is opening the Step Functions console from the AWS Management Console. Once in SFn, you can see a button to click to get started with creating a new state machine. Click it if you want to create a new SFn. This opens up a wizard, as shown in Figure 5-1.

This figure depicts first step in the process of creating a state machine. The wizard is quite busy. The left-hand menu bar shows the step we are currently in. This step allows us to either create the state machine from scratch by leaving the **Author with code snippets** radio button selected.

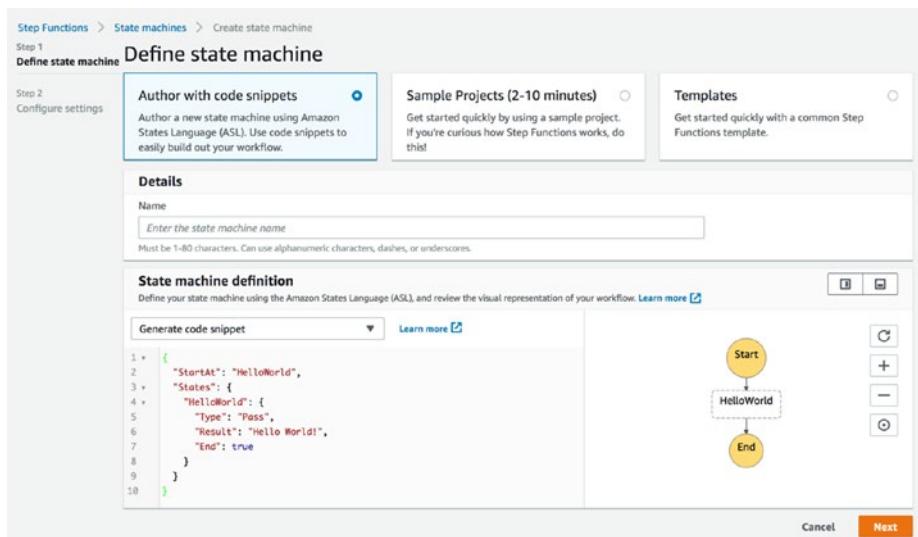


Figure 5-1. Step Functions Creation Wizard

The other two radio buttons are similar to how Lambda allows you to create from ready-made templates so that you don't have to handwrite every one of your state machine definitions. Once you give a name to your first state machine, look below the name field to find the meat of a state machine. The definition is divided into two sections.

The first section is for us to write the definition by hand, and the other half of the definition section renders the state machine visually so that we can understand it better. The visual representation helps us identify the potential issues much easier than trying to see the faults in the text. For a simple state machine like the one in the figure, it doesn't make much difference. For example, there is an issue with the following state machine. Can you identify what that is?

```
{
  "StartAt": "Send message to SNS",
  "States": {
    "Send message to SNS": {

```

CHAPTER 5 STEP FUNCTIONS

```
"Type": "Task",
"Resource": "arn:aws:states:::sns:publish",
"Parameters": {
    "Message": {
        "Input.$": "$.input.Message"
    },
    "TopicArn": "arn:aws:sns:us-west-
                2:495525968791:OrderTopic"
},
"Next": "Prep Process"
},
"Prep Process": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "Parameters": {
        "FunctionName": "arn:aws:lambda:us-west-2:495525968791:
                        function:save-order:$LATEST",
        "Payload": {
            "Input.$": "$.input.OrderPayload"
        }
    },
    "Next" : "Cooking Proccss"
},
"Cooking Process": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "Parameters": {
        "FunctionName": "arn:aws:lambda:us-west-2:495525968791:
                        function:cook-order:$LATEST",
        "Payload": {
            "Input.$": "$.input.OrderPayload"
        }
    }
}
```

```

        }
    },
    "End": true
}
}
}
}
}
```

As you are new to the state definition language (SDL), it may take some time for you to figure out what is going on with this. Even if you know SDL very well, it may be difficult for you to gather what is going on. However, if you look at Figure 5-2, you can easily tell that there is an issue of the prep process not being connected to the cooking process correctly.

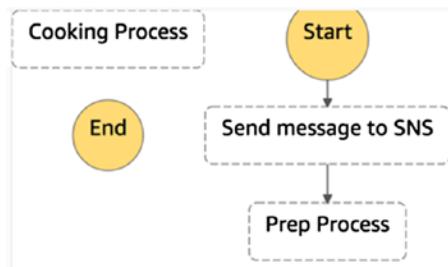


Figure 5-2. Disconnected State Machine

This can easily lead you to the error statement, which is Next attribute in Prep Process state definition, and there is a typo in the name of the next state. Instead of Cooking Process, I wrote it as Cooking Proccss. Of course, the SDL editor may have already given you an error saying that the given state doesn't exist, but sometimes an error may be logical instead of a typo. In such cases, looking at the image of the state machine helps quite a bit.

Let us look at the default state machine definition provided in the definition editor in step 1. The default state machine definition is just a hello world example of SFn. It just ends as soon as it starts. The only state in the state machine is of type Pass as you can see from the attribute type.

CHAPTER 5 STEP FUNCTIONS

A Pass state is meant for passing the input to its output without doing any work. Take a note of the default snippet in the editor. It should be somewhat similar to the following snippet:

```
{  
  "StartAt": "HelloWorld",  
  "States": {  
    "HelloWorld": {  
      "Type": "Pass",  
      "Result": "Hello World!",  
      "End": true  
    }  
  }  
}
```

As you can see, it's just passing the string "Hello World!" as its output, which means it disregards the input passed to the state machine. The StartAt attribute of the definition indicates which state is the starting point of the execution. In the HelloWorld state, we have an attribute, End, whose value is set to true. Every state should either have the Next or End attribute. Change the definition to the one shown in the following:

```
{  
  "StartAt": "HelloWorld",  
  "States": {  
    "HelloWorld": {  
      "Type": "Pass",  
      "Result": "Hello World!",  
      "Next": "Send message to SNS"  
    },  
  }  
}
```

```
"Send message to SNS": {  
    "Type": "Task",  
    "Resource": "arn:aws:states:::sns:publish",  
    "Parameters": {  
        "Message": {  
            "Input.$": "$"  
        },  
        "PhoneNumber": "+1206XXXXXX"  
    },  
    "End": true  
}  
}  
}
```

If there are more states, we can specify any one of them in the Next attribute. In our new code snippet, the Next attribute is marked as Send message to SNS, which is the name of the second state in our latest definition. This state of type Task sends an SNS notification to a phone number. You should also change the phone number to be a valid one. Once you modify the definition, click the refresh icon on the state definition image to update the visual representation.

We will go into details of task type states in later sections. For now, just click the Next button below the definition editor so we can progress to the second step. In step 2, the first configuration we need to provide is the IAM role that needs to be used for the execution of the state machine. By default, it will create a new IAM role so that it has access to the AWS services based on the state machine definition. However, there is a time lag between when the role is created and when it can be used. Let me explain. You can just click the **Create state machine** button at the bottom of the page to complete the process for now.

Executing the State Machine

Once the state machine is created, you can start execution by clicking the Start execution button in an SFn view. As soon as you click, it asks you to provide an input. Leave the default value in the pop-up, and click Start execution. The bug I specified is that if you try to start execution immediately after creating it, there is a chance that you may see an error as in Figure 5-3.

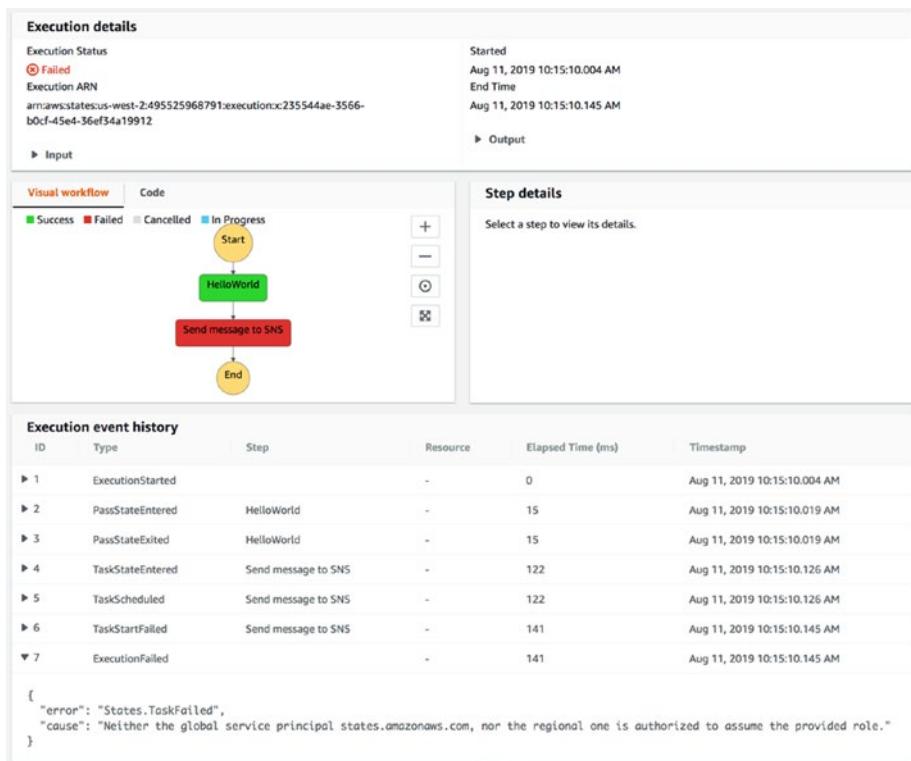


Figure 5-3. *Errored Execution*

The visual representation of an execution makes it easy to know where the issue is. The Send message to SNS is colored in red. If you are reading a black and white hard copy of this book, it may not be evident in the figure displayed on this page; however, the SFn UI should mark the SNS step in red and HelloWorld step in green.

If you want to know the actual error, scroll down to the event execution history, and expand the event with type ExecutionFailed. In our figure, it is the event with id 7. As the error states, SFn could not assume the provided role. You may get similar error if you created a role but forgot to give SNS publish access. However, this issue is only temporary. Rerun the same state machine after a minute or so. Now the execution should succeed, and you should receive an SMS to the number you have provided.

If you want to see all the executions, you can open the state machine from the list of state machines displayed in SFn. You will then be redirected to a table that shows all the executions. You can filter by running executions or passed or failed executions to quickly see what is going on with them.

API Method

If you want to create a step function using API, you need to create a step-functions client and invoke the `createStateMachine` method. This function accepts name, definition, IAM role, and tags as the input, on par with the UI. Sample code is given in the following for your reference. As you can see, it follows the same template as every other service. The first step is always to obtain credentials and create the client (Listing 5-1).

Listing 5-1. javascript-sources/chapter-5/demo-01/create_state_machine.js

```
const AWS = require('aws-sdk');

AWS.config.region = 'us-west-2';
AWS.config.credentials = new AWS.SharedIniFileCredentials
    ({profile: 'admin'});

let sfn = new AWS.StepFunctions();

let createPromise = sfn.createStateMachine({
    name: "OrderSFn",
    definition: JSON.stringify({
        "StartAt": "HelloWorld",
        "States": {
            "HelloWorld": {
                "Type": "Pass",
                "Result": "Hello World!",
                "End": true
            }
        }
    }),
    roleArn: "arn:aws:iam::495525968791:role/service-role/sfn-sns"
}).promise();

createPromise.then(data => console.log(data));
```

As this code demonstrates, we need to pass the definition as a string. We can pass the same text as we type in the definition editor. The only requirement is that we need to convert it to a string using methods such as `JSON.stringify`. The code will be cleaner this way, especially if the state machine definition has many states and branches.

In the preceding code snippet, make sure you give a role that belongs to your account. I have given a role I created so that it has access to send SNS messages from SFn. Of course, this role doesn't need to have SNS access in this example, as all I'm doing here is passing the input and nothing else. You should also make sure that you use a profile you configured in your machine. I configured an admin profile. You might have done the same if you followed the steps in the previous chapters.

To execute a state machine, we need to provide two parameters. The first one is the ARN of the state machine you want to execute, and the next important parameter is input for the execution. For our order use case, the input can be order information such as user_id, food_id, and so on.

There is another optional parameter that can be provided when starting an execution. We can give a name for the execution. It has to be unique for an AWS account, region, and state machine for 90 days. So you can use the same name for two different executions of the same state machine if they are 90 days apart or if you have created the same step function in a different region:

```
let region = 'us-west-2';
let aws_account = 595570293014;

let item = {
  user_id: "12345678" ,
  order_time: "2019-08-10 12:35:23.91",
  order_status: "Confirmed"
};

sfn.startExecution({
  name: '12345678-20190810123523.91',
  stateMachineArn: `arn:aws:states:${region}:${aws_account}:
    stateMachine:OrderSFn`,
  input: JSON.stringify(item)
}).promise().then(data => console.log(data));
```

CHAPTER 5 STEP FUNCTIONS

As you can see in the code snippet, we can pass the name. However, a name cannot contain any special characters or spaces or brackets or braces. Once we execute the preceding code, the execution will be recorded in the list of executions, as shown in Figure 5-4, associated with the step function named OrderSFn. Before you can execute this code, you should change the region and AWS account id; otherwise, the ARN formed will not match any existing step function.

| Executions (4) | | | | |
|------------------------------------|-----------|------------------------------|------------------------------|--|
| Name | Status | Started | End Time | |
| 12345678-20190810123525.91 | Succeeded | Aug 11, 2019 02:21:34.287 PM | Aug 11, 2019 02:21:34.519 PM | |
| 60880d23-e75b-425d-a0d4-b030cc02f8 | Succeeded | Aug 11, 2019 01:49:25.548 PM | Aug 11, 2019 01:49:25.564 PM | |
| 87e75f02-b525-40ec-af5c-6016ddaaeb | Succeeded | Aug 11, 2019 01:48:26.558 PM | Aug 11, 2019 01:48:26.585 PM | |

Figure 5-4. List of Executions

The previous executions have a universally unique identifier (UUID). As the name of the execution is optional, a random UUID is generated for naming the execution. If you execute same code without changing the name more than once, you will receive an “Execution Already Exists error.”

There can be many executions running in parallel for a state machine, but they cannot have the same name. The state machine executions are removed after 90 days, so once the 90-day retention period of the execution is over, a name can be reused without any issue. In our order use case, if we use user id and order time to generate an execution name, they will never be two executions with the same name.

To delete a state machine, you can use the `deleteStateMachine` method which only accepts ARN of the state machine to be deleted. Assuming that your locally configured user has access to delete state machines, it will be deleted, and an empty response is returned once the promise is fulfilled.

State Machine Definition

The previous example of a state machine gives an understanding of how to use a state machine once there is a definition. We just used the default definition of hello world to learn all the moving parts of the SFn service. However, how would you build a state machine definition from scratch? This section tries to explain the mechanisms of state definition language, SDL, so that you can build more useful state machines.

Top-Level Fields

As the earlier example made it clear, state machines are defined using JSON. Without trying to understand what each of the fields does fully, let us look at how a typical state machine definition looks like. Only the fields in bold font are mandatory. All other fields are optional:

```
{
  "Comment" : "",
  "StartAt" : "",
  "TimeoutSeconds" : ,
  "Version" : "",
  "States" : {
    "State1" : {},
    "State2" : {}
  }
}
```

Let us look at the attributes one by one to understand them better:

- Comment – Just a human readable description about the state machine being created.
- Version – Version of the state definition language. Default value is 1.0.

- **TimeoutSeconds** – An optional integer field to specify how long should the state machine run. If the execution is not completed in the given number of seconds, the execution is marked failed with a `States.Timeout` error.
- **States** – This field represents various states the machine can be in. There must at least be one state in each state machine. We will go into how to create the states in the next section.
- **StartAt** – As a state machine can have more than one state defined in the `states` field, this field specifies the first state of the execution. Once the SFn receives the command to start execution, it will set the state of the machine to be the one mentioned in the `StartAt` field.

Every state machine, however complex, follows the same pattern of definition. The only thing that changes is the number of states and their transitions from one to another till the state machine reaches a terminal state. States can be simple ones like `Pass` as we have seen before or branches like `choice` or `parallel`. The order in which the states are defined in the `states` field doesn't influence the order in which they get executed.

States

Earlier, I have defined what is a state in simple terms, and in the example, we have seen a state of type `Pass` and of type `Task` while creating a state machine. However, we haven't seen much of the details of how these states can be used and what other types of states can there be. As the creation of a state machine is purely done by defining various states, we have to learn details of the extent to which we can use them.

As you have also seen, a state machine can express an algorithm employing one or more states defined through its definition. Individual states can make decisions, perform tasks based on inputs given to them, and pass the output to other states.

Let us touch base with the ones we have already covered and see different facets of their use cases. However, before we do so, here are different kinds of states there can be as of writing this book. Each state type is intended for a specific function in state machine:

- Pass: Simply pass input to its output and optionally inject additional data which is fixed while defining the state machine.
- Wait: Delay execution of one or more states for a fixed period or until given date/time.
- Task: Perform some work associated with the state machine. It can be the execution of a lambda function or sending a message to a queue or even updating a DynamoDB item.
- Choice: Allows us to choose between two or more branches in the algorithm of the state machine. We have earlier seen an example of initiating a refund activity. If the amount of the order is more than 15 dollars, a Choice state allows us to decide whether to execute that refund branch or not.
- Parallel: Initiates simultaneous execution of multiple branches.
- Fail: Stop the execution with a failure status.
- Succeed: Stop the execution with a success status.

Common State Fields

Each state has a set of fields it accepts along with common fields that are applicable for all state types. So before we go into each state type, let us quickly see what they are:

- Type (Mandatory) – To specify the type of the state.
- Next (Conditional) – This field specifies the next state to run if the current state finishes. This field is mandatory if we don't specify the field "End."
- End (Conditional) – If a state has this field with a value of true, the state is considered as the terminal state of the state machine. The Choice state doesn't support this field. A state machine can have more than one terminal state, but each state can either have the Next or End field specified.
- Comment (Optional) – Human-readable description of the state.
- InputPath (Optional) – Holds a JSON path to select a portion of the state's input for task processing. Default value is \$, which means the entire input is used by the task.
- OutputPath (Optional) – A JSON path to specify a portion of the input that should be passed to the output. Default value is \$, so the entire input is passed on to the output of the state.

Pass State

As we have seen what these different types of states do, let us now learn them in detail one by one, beginning with the Pass state. A Pass state (“Type”: “Pass”) passes the input to its output without performing any work. However, it can also be used just to inject some fixed data while passing the input. In addition to the common fields, a Pass type of state accepts the following fields:

- Result – Used to specify the output of the Pass state to be passed to the next state. It is often used in conjunction with the ResultPath field.
- ResultPath – This is used to merge the input of the state with the values provided in the Result field.
- Parameters – Create a key-value pair that will be passed as input. We can either use constants, or it can be picked from the input of the state.

Manipulate Input and Output

To understand this better, let us look at the examples we have seen earlier. Here is the HelloWorld state without the Result field. As the Result field is optional, we can ignore it. In such cases, the input will be passed as output. Open the execution and click the HelloWorld state and expand the Input and Output sections, as shown in Figure 5-5.

```
"HelloWorld": {
    "Type": "Pass",
    "Next": "Send message to SNS"
}
```



Figure 5-5. Input and Output Passthrough

Result

The input and output are the default values provided when we try to execute a state machine. The output of the HelloWorld state is passed as is, to send a message to SNS. Let us now see how to overwrite the input with a static string. In fact, you have already seen this example, as shown in Figure 5-6.

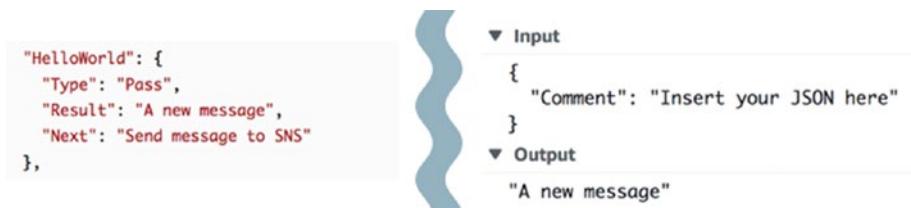


Figure 5-6. Using the Result Field Overwrites the Output

ResultPath

The previous two methods will either entirely overwrite the input to generate new output or don't touch it at all. By using the ResultPath parameter, we can control where the value in the Result field should be injected in the input as shown in Figure 5-7.

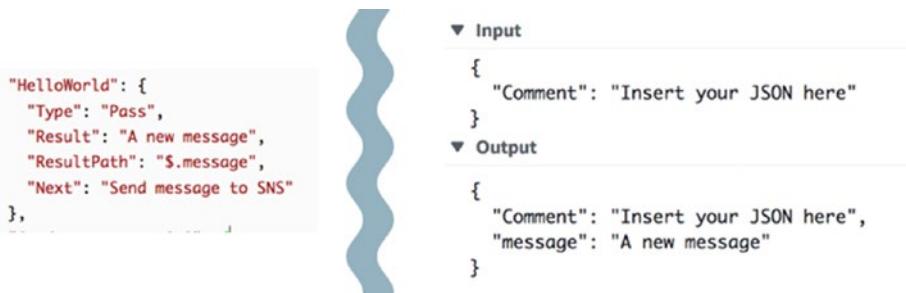


Figure 5-7. Combining Input with Additional Fields

You can pass a JSON as value for Result if you want to use complex objects instead of a string. Just don't enclose that in double quotes. If you enclose it in double quotes, it is passed as a string, so it can't be converted to JSON quickly. Here you can find an example snippet and the resulting output from the task:

```

"HelloWorld": {
  "Type": "Pass",
  "Result": {
    "addon": "A new addon info"
  },
  "ResultPath": "$.message",
  "Next": "Send message to SNS"
}
{
  "Comment": "Insert your JSON here",
  "message": {
    "addon": "A new addon info"
  }
}
  
```

The value in ResultPath is a JSON path to indicate the JSON field that should get the value of the Result attribute. In the JSON path, \$ indicates the root, so the path \$.message indicates a message attribute at the root of the JSON. If you want to add message as a child to another attribute named addon, which is at the root level, the JSON path would be \$.addon.message. JSON path has many other features to specify more complex paths, but that would eat up a lot of pages here. Feel free to go to the following GitHub repository to understand it better:

<https://github.com/json-path/JsonPath>

Parameters

The final field in the Pass state, Parameters, is used to create key-value pairs from the values that are passed in as input and mix them with few static values. We can also fetch values from the execution context. Assume that we have the following input:

```
{  
    "user_id": "1234567890",  
    "order_time": "2019-08-10 15:45:12.101",  
    "total_amount": 10.5  
}
```

If you want to create a key-value pair of total amount to send to the user as an SMS, the message can only have information about the amount he has to pay instead of internal details such as user id. Here is an example of how to create key-value pairs:

```
"HelloWorld": {  
    "Type": "Pass",  
    "Parameters": {  
        "comment": "Your order is received",
```

```

    "OrderDetails": {
        "amount.$": "$.total_amount"
    }
},
"Next": "Send message to SNS"
}

```

In this example, we have created a mix of static information, “Your order is received” and total amount. The static information is no different from what you have seen so far, except that you have to mention it in the Parameters field. The field you want to extract from the input is provided as `$.total_amount`. Every key that needs to get value from input has to be suffixed with a `“.$”`.

The context object contains information about execution. It is an internal JSON that is available during execution. It includes information about the state machine and execution. The context object has the following information:

```

{
    "Execution": {
        "Id": "String",
        "Input": {},
        "StartTime": "Format: ISO 8601"
    },
    "State": {
        "EnteredTime": "Format: ISO 8601",
        "Name": "String",
        "RetryCount": Number
    },
    "StateMachine": {
        "Id": "String"
    },
}

```

```
"Task": {  
    "Token": "String"  
}  
}
```

If you want to fetch information such as state machine execution start time or retry count of the current state to make decisions about what to do next as a part of current retry, prefix the JSON path with a \$\$ as shown in the following snippet:

```
"Parameters": {  
    "comment": "Your order is received",  
    "retry_count.$": "$$.State.RetryCount",  
    "OrderDetails": {  
        "amount.$": "$.total_amount"  
    }  
}
```

However, if you execute this as it is, it may fail to fetch the retry count because retry count only exists for states that have failed and are being retried. So far, we have not seen how to configure retries. You can use other information that exists all the time, such as state entered time, without any issues from the first attempt. If the parameter couldn't be found, you may receive an error such as this:

```
An error occurred while executing the state 'HelloWorld'  
(entered at the event id #2). The JSONPath '$$.State.  
RetryCount' specified for the field 'retry_count.$' could not  
be found in the input
```

Wait State

A Wait state (“Type”: “Wait”) allows you to pause the execution of the state machine. Oftentimes, you may need to wait for a few dependencies to be satisfied before you can make progress. Our state machine may need a file to process, and that may be generated every day at midnight. If we reach the file process state before midnight, we will have to make the execution pause until that point in time. In other words, we use absolute time.

There may be other scenarios where we may want to wait for a fixed amount of time such as when the order is being prepped, we wait for 10 minutes or so before probing to find out if the order preparation is successful. In such cases where we need to wait for a certain period of time or till a time point is reached, we can use a Wait state to suspend the workflow execution temporarily. We use relative time in these cases.

In addition to common fields, the Wait state accepts the following fields:

- Seconds – Time in seconds to wait before continuing the execution.
- Timestamp – An absolute point in time in ISO 8601 format until which the execution is paused.
- SecondsPath – Time in seconds but instead of a constant value, it is the JSON path so that this seconds value can be taken from the input provided to the step.
- TimestampPath – JSON path of the field that has an absolute point in time until which the execution is to be held in the current state.

The purpose of all these fields is same, to specify when the execution can proceed to the next state. However, only one of these fields can be provided at any time. Here are a few examples of how to use a Wait state. Using any of these examples shown in Table 5-1, the execution will wait until the wait step is completed.

Table 5-1. Sample usages to pause a state machine execution

| Static | Variable |
|---|--|
| <pre>"five_minute_wait": { "Type": "Wait", "Seconds": 300, "Next": "Cook" }</pre> | <pre>"absolute_wait" : { "Type": "Wait", "SecondsPath": ".\$.delay", "Next": "NextState" }</pre> |
| <pre>"absolute_wait" : { "Type": "Wait", "Timestamp": "2016-03-14T01:59:00Z", "Next": "NextState" }</pre> | <pre>"absolute_wait" : { "Type": "Wait", "TimestampPath": ".\$.startTime", "Next": "NextState" }</pre> |

When the state machine is in a Wait state, you can notice that on the visual representation of the workflow execution with ease as the wait task will be in blue for the given number of seconds as shown in Figure 5-8. After waiting for the number of seconds, you can refresh the page to observe that the state machine moved past the wait.

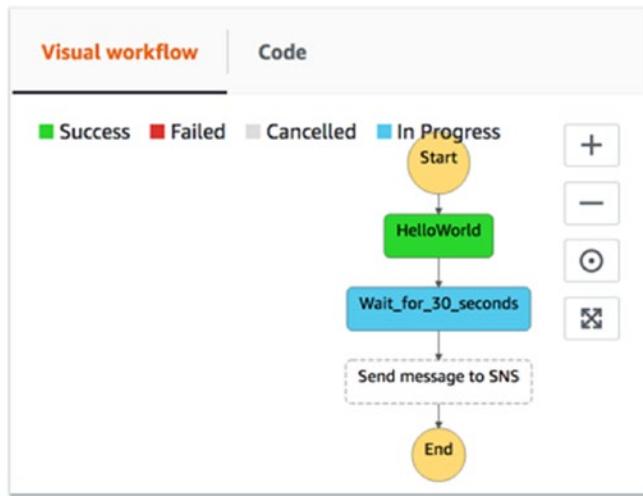


Figure 5-8. Execution Pausing at the Wait State

Task State

A Task state (“Type”: “Task”) is the most versatile state available in the state definition language. In simple terms, a Task state is a single unit of work performed by a state machine. A task can be a state that sends an SMS using SNS or a Lambda function that copies tens of small files from one S3 bucket to another or an activity implemented and runs in EC2 hosts.

Basically, tasks are the states where bulk of the work happens in every state machine execution. Without tasks, a state function is nothing but a decorator that just transforms the data from one format to another and displays it – no useful application runs on this alone. A Task state can accept the following fields along with the common fields.

Resource

These mandatory fields specify where the work will be performed. It can be a Lambda or activity or any other supported AWS service. It is just ARN of the entity that performs the task. They follow a similar pattern of other ARNs you may have seen so far. Here is the complete pattern:

```
arn:partition:service:region:account:task_type:name
```

- Partition – AWS Step Functions partitions to use, typically AWS.
- Service – It should either be lambda for lambda type of tasks, or for AWS services or activities, this must be states.
- Region – Region in which the resource is located.
- Account – Account to which the resource belongs.
- Task type – This can either be an activity, or functions to indicate lambda functions or name of the AWS service.
- Name – Either name of the lambda function or API name.

Here are a few examples of resource values. If the region and account number are not provided, the same will be used from the step function. These examples indicate a lambda function, an SNS publish operation, an activity, and a DynamoDB put item operation, respectively:

```
arn:aws:lambda:us-west-2:1234567890:function:PrepOrderLambda  
arn:aws:states:::sns:publish  
arn:aws:lambda:us-west-2:1234567890:activity:PrepOrderActivity  
arn:aws:states:::dynamodb:putItem
```

Parameters

Used to pass information required by the resource. For example, an SNS resource may need a phone number and a message that needs to be sent to the phone number. This is an optional field. Here is an example of how parameters can be used. The parameters change based on the AWS service. You have already seen an example of parameters, when you created a state machine that sends an SMS. Here are the examples (Table 5-2) of how the parameters look like for different tasks.

Table 5-2. Parameters based on the target service

| Operation | Parameters |
|---------------------|--|
| DynamoDB put item | <pre>"Parameters": { "TableName": "feedback", "Item": { "Column": { "S": "MyEntry" } } }</pre> |
| SQS publish message | <pre>"Parameters": { "QueueUrl": "https://sqs.REGION.amazonaws.com/ACCOUNT_ID/myQueue", "MessageBody": { "Input": "Hello from Step Functions!" } }</pre> |

(continued)

Table 5-2. (continued)

| Operation | Parameters |
|------------------------|--|
| Invoke Lambda function | <pre>"Parameters": { "FunctionName": "arn:aws:lambda::: function:FUNCTION_NAME", "Payload": { "Input.\$": "\$" } }</pre> |

Retry

This is an optional field that consists of multiple retrier objects. These objects define retry policies in case the state execution results in any error. Each retrier represents a certain number of retries with increased timeouts between each successive failure. Each retry policy contains the following fields:

- ErrorEquals – An array of strings that match error names. If executing the state returns an error, it will be matched against the array of strings provided in this field. If any of them is equal to the error returned by the task, the retry policy will be applied. Here are a few examples:
 - States.ALL – Matches every error
 - States.Timeout - If the execution of the state timesout
 - java.lang.IllegalStateException - An exception thrown by a Java application
- IntervalSeconds – Number of seconds to wait before retrying for the first time. Default value is 1.

- MaxAttempts – A number representing how many times the state will be retried after successive failures. The default value is 3. If you provide 0, the task will never be retried.
- BackoffRate – A multiplier to increase the interval seconds every time the state fails. If the initial interval seconds is 1 and the backoff rate is 1.5, when the state fails for the first time, the gap between successive attempts is 1 second, and if the state function fails while retrying, it waits for 1.5 seconds. If the task fails again, the interval seconds will become $1.5 * 1.5 = 2.25$ seconds. If the max retries are more than two and if the state fails again, the next interval would be 3.375 seconds.

Here is an example of a simple Retry field with multiple retrier objects:

```
"Retry": [  
  {  
    "ErrorEquals": [ "States.Timeout", "java.lang.  
                    FileNotFoundException" ],  
    "IntervalSeconds": 10,  
    "BackoffRate": 2.0,  
    "MaxAttempts": 3  
  },  
  {  
    "ErrorEquals": [ "com.fasterxml.jackson.core.  
                    JsonParseException" ],  
    "MaxAttempts": 0  
  },
```

```
{
  "ErrorEquals": [ "States.ALL" ],
  "IntervalSeconds": 5
}
]
```

In this example, if the state times out or if a file the state expecting is not found, we are retrying after 10 seconds, 20 seconds, and 40 seconds if the state keeps failing. After that, the state will be considered as fail. In case there is an issue such as `JsonParseException`, which may be something that cannot be recovered from or, in other words, however many times we retry, it may still end up with the same exception, so we are not retrying.

In case of any other error, we retry once after 5 seconds of interval because the default value for the max attempts field is 1. If the state machine transitions out of the current state for any reason, the retry parameters are reset. If the execution of the state succeeded on the second attempt after receiving `FileNotFoundException` once and moves out of the current state and reenters the same state because of the way the state machine is defined, the previous counter of how many times this state has been retried is reset to zero.

Catch

Similar to `Retry`, the field `Catch` comes into play in case of a state failure. The value of the `Catch` field contains an array of objects called catchers. When the state fails even after all the retries are exhausted or if there were no retries to begin with, SFn iterates through all catchers to find a matching one.

The matching catcher helps the SFn to direct the state machine execution to another state. In the absence of catcher objects, the state machine execution is marked as failed. Each catcher object can contain the following fields to help SFn to direct the execution appropriately:

- `ErrorEquals` – An array of error names including `States.ALL` to match all error names
- `ResultPath` – To append the error result to input before passing on as output of this state
- `Next` – To specify the name of the state to which the state machine should transition to

Here is an example of a typical `Catch` field. Can you understand what it is doing?

```
"Catch": [
  {
    "ErrorEquals": [ "com.fasterxml.jackson.core.JsonParseException" ],
    "ResultPath": "$.error-info",
    "Next": "CreateInputJson"
  },
  {
    "ErrorEquals": [ "States.ALL" ],
    "Next": "EndState"
  }
]
```

In this example, if we get a `JsonParseException`, we want to recreate input JSON. So we are transitioning the state machine to a state named `CreateInputJson`. Assuming that we have a state that creates this file, state machine execution continues further or in a loop depending on how we have created this. We can use the `Catch` field in combination with `Retry`, as shown in the following:

```
"Retry": [
  {
    "ErrorEquals": [ "ErrorA", "ErrorB" ],
    "IntervalSeconds": 1,
```

```

    "BackoffRate": 2,
    "MaxAttempts": 2
},
{
  "ErrorEquals": ["ErrorC"],
  "IntervalSeconds": 5
}
],
"Catch": [
{
  "ErrorEquals": ["States.ALL"],
  "Next": "Z"
}
]

```

In this code snippet, the state will be directed to Z if ErrorC occurs once or if ErrorA or ErrorB occurs three times in any combination. So if we get ErrorA for the first time, ErrorB for the second time and ErrorA again for the third time in same retry loop, it is still considered as 3 successive failures matching first retrier object.

Remember that the counter is reset if the state transitions to Z irrespective of what error is received. So SFn doesn't remember about how many times we got ErrorA or ErrorB once we applied the catcher policy and moved to Z. The counters retain their values as long as they are in a single retry loop.

TimeoutSeconds and HeartbeatSeconds

Timeouts are a common phenomenon in a distributed environment. Depending on how the tasks are created, they may take anywhere from a few seconds to a few days. However, it is essential for us to recognize when the task is making progress and when there is no more hope of succeeding.

If a task of copying a file from one S3 bucket to the next usually takes 10 seconds, there is no point of waiting any longer than 10 seconds in an ideal world. However, delays can happen; the file we are supposed to copy has not been generated yet. So we need to provide a leeway for our task to wait. We need to decide on how long we can wait for the task to complete. It all depends on your business. Maybe 5 minutes is okay, or a few hours may be fine as well.

In such cases where we want the SFn to wait for a task to be completed in a certain amount of time, we specify `TimeoutSeconds`. If a task is not completed within that time, we get a `States.Timeout` error thrown; and if there is a retrier object matching with that error, the task will be retried as per policy.

In long-running tasks, it is wise to keep a regular check of whether our task is progressing or not. We expect our activity or lambda or AWS service to communicate that the work is being carried out by regularly sending heartbeat. For specifying the maximum permissible interval between two successive heartbeats, we use the `HeartbeatSeconds` field.

ResultPath

This is an optional field to specify where the results received from resource should be placed in the output of this state. This is used to inject result into input before sending out as the output of the state. You can use JSON path to specify a path. This is similar to how `ResultPath` works with the `Pass` state.

With all these parameters in place, a state may look much complex than it is. Here is an example of a typical state that runs a Glue job:

```
"Start Glue Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::glue:startJobRun",
    "Parameters": {
        "JobName": "S3ToRedshift"
    },
}
```

```
"Next": "DeleteInputFiles",
"TimeoutSeconds": 300,
"HeartbeatSeconds": 60,
"Retry": [
  {
    "ErrorEquals": [ "ErrorA", "ErrorB" ],
    "IntervalSeconds": 1,
    "BackoffRate": 2,
    "MaxAttempts": 2
  }
],
"Catch": [
  {
    "ErrorEquals": [ "States.ALL" ],
    "Next": "FailureState"
  }
]
}
```

Succeed and Fail States

Both these states are intended to end the execution of a workflow, albeit with different status codes. A Succeed state (“Type”: “Succeed”) ends the execution with a successful status code. In a quite opposite manner, a Fail state (“Type”: “Fail”) ends the execution in an errored state. None of these two have the Next or End field.

However, the Fail state has few additional fields to specify the cause of failure and an error message. The fields for each of them, respectively, are Cause and Error. Both these fields are optional. You can find examples of how to use these states in Table 5-3.

Table 5-3. Examples of Succeed and Fail states

| Succeed State | Fail State |
|---|--|
| <pre>"SuccessfulState": { "Type": "Succeed" }</pre> | <pre>"FailureState": { "Type": "Fail", "Cause": "Missing S3 file", "Error": FileNotFound }</pre> |

Choice State

A Choice state (“Type”: “Choice”) adds the ability to branch between states. In regular non-JSON programming terminology, it is equivalent to If-Else or switch case. A Choice state contains the following two fields at the top level apart from common state fields:

- Choices – Array of choice rules to determine which state the state machine should transition to.
- Default – This is an optional field that specifies which state the state machine should transition to if none of the choice conditions are met. With these two fields, a Choice state looks like the following snippet:

```
{
  "Type": "Choice",
  "Choices" : [
    {
      // choice rule 1
      "Next": "StateX"
    },
  ]}
```

```
{
    // choice rule 2
    "Next": "StateY"
}
],
"Default": "StateZ"
}
```

Choice Rules

A choice rule is a condition that resolves into a Boolean value. If the value is true, the Choice state transitions the state machine into the state specified in its Next field. If the condition results in a false, the rest of the choice rules are processed in the order they appear in the state machine definition.

A choice rule consists of two fields. The first one is an input variable, and the other is a comparison operator. Let us look at an example to understand it better:

```
{
  "Variable": ".$.order_state",
  "StringEquals": "PREPPED",
  "Next": "CookState"
}
```

In this example, we are assuming that the input for this Choice state has an order_state attribute. If the value of that attribute is equal to PREPPED, we know that the order is ready and it has to be cooked now. So we transition the state machine to CookState. Instead of StringEquals, we can use many other operators like the ones shown in Table 5-4.

Table 5-4. Various types of operations available

| Numeric Operators | String Operators | Boolean and Timestamp Operators |
|---------------------------|--------------------------|---------------------------------|
| NumericEquals | StringEquals | BooleanEquals |
| NumericGreaterThan | StringGreaterThan | TimestampEquals |
| NumericGreaterThanOrEqual | StringGreaterThanOrEqual | TimestampGreaterThan |
| NumericLessThan | StringLessThan | TimestampGreaterThanOrEqual |
| NumericLessThanOrEqual | StringLessThanOrEqual | TimestampLessThan |
| | | TimestampLessThanOrEqual |

These operators are pretty easy to understand what they are doing. They are equivalent to mathematical symbols ==, >, >=, <, and <= for each data type. The Boolean data type doesn't have greater than or less than comparisons for obvious reasons. For each of these operators, the corresponding variable value must be of the appropriate type.

There are a few exceptions and limitations. Usually, a timestamp is represented as a string, so it is possible to use string operations in timestamp variable values. The timestamp must conform to RFC3339 profile ISO 8601, and a 'T' should always separate the time and date portions and a 'Z' must be present if the time zone offset is not present. String comparison follows the same rules as the compareTo method of Java, and integers should be in the range [1-2^53, -1+2^53]. If the numbers fall outside of this range, the comparison results cannot be predicted in a foolproof manner.

The Choice state may be too difficult to follow in its text form, but the visual rendering of the same will keep things very clear. Have a look at the state definition and its associated visual representation in Figure 5-9.

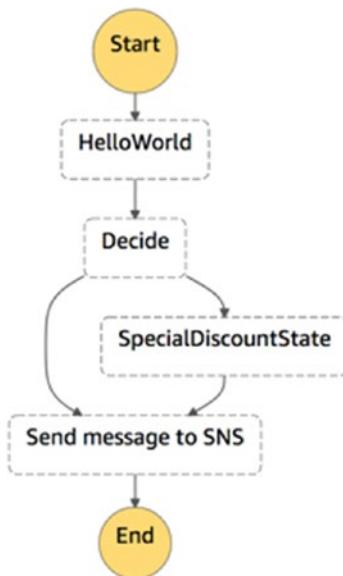


Figure 5-9. Visual Rendition of Choice

```
{  
  "StartAt": "HelloWorld",  
  "States": {  
    "HelloWorld": {  
      "Type": "Pass",  
      "Next": "Decide"  
    },  
    "Decide": {  
      "Type": "Choice",  
      "Choices" : [  
        {  
          "Variable" : "$.time_difference",  
          "NumericGreaterThanOrEqualTo": 3600,  
          "Next": "SpecialDiscountState"  
        }  
      ]  
    }  
  }  
}
```

```

    ],
    "Default": "Send message to SNS"
},
"Send message to SNS": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
        "Message": {
            "Input.$": "$"
        },
        "PhoneNumber": "+1xxxxxxxxxx"
    },
    "End": true
},
"SpecialDiscountState" : {
    "Type": "Pass",
    "Next": "Send message to SNS"
}
}
}
}

```

In this example, we are executing a decision state soon after the HelloWorld state. The decision state may directly send the flow to the state that sends a message to SNS, or it may transition the machine to a special discount state which will guide the state machine to a send message state.

Combining and Negating Choice Rules

The previous section gives a complete list of conditional operators. In our applications we code in day-to-day life, we may have to combine multiple conditions. We may only want to execute a refund state if the order amount is more than \$50 and if the total time to deliver is more than 1 hour.

CHAPTER 5 STEP FUNCTIONS

In other cases, the condition is an either-or type. We may want to give a special discount if the order amount is more than \$50 or if the delivery is delayed by more than 1 hour. This is a slight modification to our earlier condition. In either case, we can't do it with just one condition. So we need to combine multiple conditions using And/Or operators. For either of them, the syntax looks like this:

```
{  
  "And" : [  
    {  
      "Variable" : "$.total_amount",  
      "NumericGreaterThanOrEqual": 50  
    },  
    {  
      "Variable" : "$.time_difference",  
      "NumericGreaterThanOrEqual": 3600  
    }  
,  
  "Next" : "SpecialDiscountState"  
}
```

This is an example of how to write a choice rule to combine two different choice rules. We can use “Or” instead of “And” to change the condition from both should satisfy to one of them should satisfy. We can nest them as well for building even more complicated choice conditions as shown in the following snippet:

```
{  
  "Or": [  
    {  
      "And": [  
        {  
          "Variable": "$.total_amount",  
          "NumericGreaterThanOrEqual": 50  
        },  
        {  
          "Variable": "$.time_difference",  
          "NumericGreaterThanOrEqual": 3600  
        }  
      ]  
    }  
  ]  
}
```

```

        "NumericGreaterThanOrEqual": 50
    },
    {
        "Variable": "$.time_difference",
        "NumericGreaterThanOrEqual": 3600
    }
]
},
{
    "Variable": "$.total_amount",
    "NumericGreaterThanOrEqual": 100
}
],
{
    "Next": "SpecialDiscountState"
}

```

In this extended example, we specify that either “if the order amount is \$50 and the time difference is 1 hour” or “if the order amount is 100”, set the machine’s state to special discount. We can use these to fulfill complex business requirements of the day.

We can also negate a choice rule by using the Not operator. The AWS state definition language doesn’t provide a StringNotEquals or NumericNotEquals. Instead, it allows us to use the Not operator to achieve the same effect. Here is an example of how to use a Not operator to inverse the result of a string equals condition:

```

{
    "Not": {
        "Variable": "$.order_state",
        "StringEquals": "Cancelled"
    },
    "Next": "NextState"
}

```

As usual, this can also be combined along with other choice rules and operators to inverse the result. By combining these Choice states, we can build state machines that are quite complex.

Parallel State

Let us rehash the previous scenario of Choice state. We execute a special discount state if the input matches a choice. However, we will also execute a send message state after executing a special discount state. However, there is no need for us to execute both states sequentially. We can execute them in parallel as long as there is no functional dependency.

To execute two states in parallel, we can make use of the Parallel state to combine them. Just like the Choice state, a Parallel state is used to create branches of execution in the state machine. Along with common fields, the Parallel state has Branches, ResultPath, Retry, and Catch fields.

The Branches field is an array of state machines that must be executed in parallel. Each state machine needs to contain StartAt and States fields like any other top-level state machine. The ResultPath is to specify where the output of the branches should be injected into input before passing on as Parallel state's output.

We have already seen catchers and retriers in the Task state. They don't behave any different in this state. Here is a simple example of how a Parallel state can be defined. I have ignored all optional fields to reduce complexity as you know what they are anyways. Here is the definition and visualization, as shown in Figure 5-10, side by side.

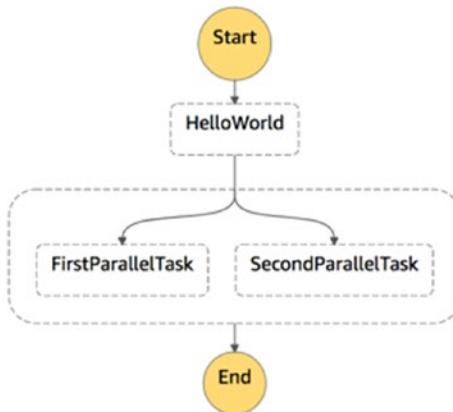


Figure 5-10. Parallel Tasks

```
{  
  "StartAt": "ParallelState",  
  "States": {  
    "ParallelState": {  
      "Type": "Parallel",  
      "End": true,  
      "Branches": [  
        {  
          "StartAt": "FirstParallelTask",  
          "States": {  
            "FirstParallelTask" : {  
              "Type" : "Pass",  
              "End" : true  
            }  
          }  
        },  
        {  
          "StartAt": "SecondParallelTask",  
          "States": {  
            "SecondParallelTask" : {  
              "Type" : "Pass",  
              "End" : true  
            }  
          }  
        }  
      ]  
    }  
  }  
}
```

CHAPTER 5 STEP FUNCTIONS

```
        "SecondParallelTask" : {  
            "Type" : "Pass",  
            "End" : true  
        }  
    }  
}  
]  
}  
}
```

A Parallel state executes all branches as concurrently as possible and waits until all the branches finish execution. Any output returned by the branches is accumulated in an array and passed on to the next stage. So if the input to this state machine is the default JSON provided by the step function execution wizard, the output will have it doubled as shown in the following:

```
{ "Comment": "Input" } → [ { "Comment": "Input" }, { "Comment": "Input" } ]
```

The reason for this duplication is an accumulation of outputs from each of the Parallel states into an array. Both our Parallel states are of type Pass. They are not doing anything to the input before passing the same as output. We can use ResultPath to assign this array to another attribute, but you can't get rid of the array. Usual rules are applicable even for sub-state machines inside the Parallel state.

Activities

The previous chapters demonstrated how to invoke various services of AWS like SNS, Lambda, and so on. However, there can be times where we don't want to create a Lambda worker for a specific state for various reasons. Sometimes an activity has to be performed in the mobile device of a customer using our application. Alternatively, a chef has an Alexa device where he asks for what he has to cook as soon as he is available.

In such cases, we only want the code to be executed when the user of the code requests it to be executed, and the code may have to run anywhere the client needs. In Step Functions, activities are a mechanism to link code running anywhere with a task in a state machine. Let us see how we can create and use activities to link a state machine execution with a program running on our machine.

Creating and Using Activity

Creating an activity can either be done from UI or by using API. In either case, it is a very simple process, and we will receive an ARN that can be used to poll for pending activities. Let us see both approaches. In the SFn console, you will find a link named Activities underneath the Step Functions link in the left-side navigation bar, as shown in Figure 5-11.

The wizard replaces the SFn UI on the same page. You just have to give a name for the activity. Tags are optional and are used for cost exploration purposes. Provide the name and click the Create activity button located at the bottom of the wizard. Once done, the next page contains ARN details that should be as shown here:

```
arn:aws:states:us-west-2:XXXXXXXXXX:activity:prep_order_activity
```

CHAPTER 5 STEP FUNCTIONS

The name of the activity is present in the ARN as well. This ARN can be used by activity workers to pull for activities waiting to be performed. The ARN contains the keyword activity to distinguish it from other resources in AWS.

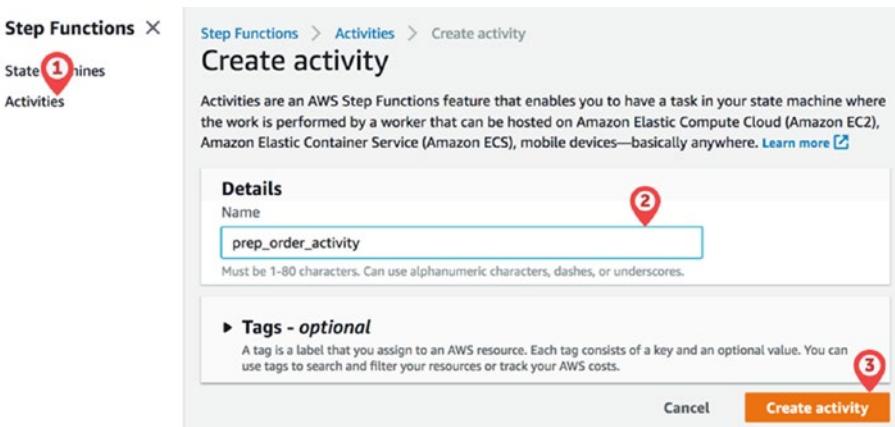


Figure 5-11. Creating an Activity from SFn

Creating the activity using API is as simple as shown in Figure 5-11. We just have to invoke `createActivity` API on the SFn client object. We should pass the name property as the input for this activity. We can also pass tag information like we did in the SFn console. The method returns ARN of the activity that has been created. If we invoke the same API twice, we get the same ARN back:

```
let sfn = new AWS.StepFunctions();

sfn.createActivity({
  name: "prep_order_activity"
}).promise().then(
  data => console.log(data)
);
```

I have already specified how to use the activity in a state. We use it in a state of type Task and provide the ARN generated by the create activity step. Here is an example of how the activity is used in a state machine:

```
"PrepareOrder": {  
    "Type": "Task",  
    "Resource": "arn:aws:states::::activity:prep_order_activity",  
    "Next": "CookOrder"  
}
```

This is just an example of the state that waits for this activity to be triggered and times out if no activity worker picks up the tasks and finishes before one of the timeouts occurs. Either a heartbeat timeout or state timeout or state machine execution timeout can fail the entire execution if the worker doesn't respond in time. Let us now build a simple activity worker that can pick from where the state machine paused.

Activity Worker Implementation

If you have gone through Chapter 3 and written an activity or decision task worker, you are already familiar with the template of this section. A worker has to poll using GetActivityTask API for the activity that it is responsible for. Once the poller receives a task that has been queued, it can work on it. It can then return the result using anyone of the following APIs:

- SendTaskSuccess
- SendTaskFailure
- SendTaskHeartbeat

CHAPTER 5 STEP FUNCTIONS

The following snippet of code fetches any activity tasks pending to be worked upon. Replace the XXXXXXX with your account number, and the region should also be based on your configuration:

```
sfn.getActivityTask({
  activityArn: "arn:aws:states:us-west-2:XXXXXXXX:activity:prep_
    order_activity"
}).promise().then(data => console.log(data));
```

This code snippet will simply log a response like the one shown here. The taskToken is an entry maintained by step functions to recognize the response of a specific task in a gazillion of activities it keeps track of, at any point in time:

```
{
  taskToken: 'AAAAKgAAAAIAAAAAAAAAAWEBT..... 3m29TIg1aZgw=',
  input: '{\n    "Comment": "Insert your JSON here"\n}'
}
```

We will also receive the input of the state through the input attribute. We need to parse it and assign it to an object if it contains a valid JSON; otherwise, you can use the input directly as a string. As soon as a worker receives the activity, SFn records the event as activity execution started, as shown in Figure 5-12.

| # | Event | Step Function | Execution ID | Timestamp |
|---|-------------------|---------------|--------------|---------------------------------|
| ▼ 3 | ActivityScheduled | PrepareOrder | - | 17 Aug 21, 2019 08:46:07.566 PM |
| { | | | | |
| "resource": "arn:aws:states:us-west-2:495525968791:activity:prep_order_activity", | | | | |
| "input": { | | | | |
| "Comment": "Insert your JSON here" | | | | |
| }, | | | | |
| "timeoutInSeconds": null, | | | | |
| "heartbeatInSeconds": null | | | | |
| } | | | | |
| ▼ 4 | ActivityStarted | PrepareOrder | - | 82 Aug 21, 2019 08:46:07.631 PM |
| { | | | | |
| "workerName": null | | | | |
| } | | | | |
| ▼ 5 | ExecutionTimedOut | - | 300001 | Aug 21, 2019 08:51:07.550 PM |
| { | | | | |
| "error": "States.Timeout", | | | | |
| "cause": null | | | | |
| } | | | | |

Figure 5-12. Activity-Related Events

The figure shows three events associated with the activity. The first one with event id 3 is raised when the SFn decides that it is time to start executing the activity and the activity is scheduled. The first event contains input to be passed to the worker, number of seconds the state machine execution should wait for total execution, and how long it should wait until you hear a heartbeat from the worker.

The event `ActivityStarted` is recorded once the worker initiates the `getActivityTask` call. When a worker makes an API call to get the tasks, we can also send a worker name in the same call using the `workerName` attribute along with `activityARN`. As our code didn't send any worker name, the name in the event in the screenshot is null. It is a good practice to send the worker name to keep a log.

The final event in the screenshot is what happens when we don't finish working on the data within the time period configured when defining the state machine. When the execution times out, the execution is failed with an error state of `States.Timeout` as you can see in Figure 5-12.

The worker can perform any activity. If the prep activity is supposed to show the list of ingredients to be collected for the given food to be cooked in a tablet where our worker is running, it does so. Once the assistant chef gathers the items one by one, he keeps them ready for the next activity and clicks the `Finish` button presented on the same screen. It may take an arbitrary amount of time. To send the completion event, we can use one of the APIs mentioned previously. Here are the example snippets to send task success or failure, as shown in Listing 5-2.

Listing 5-2. `javascript-sources/chapter-5/demo-02/activity_worker.js`

```
let sfn = new AWS.StepFunctions();

function processData(data) {
  let order = JSON.parse(data.input);
  let success = false;
```

CHAPTER 5 STEP FUNCTIONS

```
// some complex work that may take a lot of time to finish
// this process should ideally set the success variable to true
if(success === true) {
    order.status = "PREPPED";
    sfn.sendTaskSuccess({
        taskToken: data.taskToken,
        output: JSON.stringify(order)
    }).promise()
        .then(successResult => console.log(successResult))
        .catch(error => console.log(error));
} else {
    console.log("Error preparing the order");
    sfn.sendTaskFailure({
        taskToken: data.taskToken,
        error: "ERROR107",
        cause: "Error Preparing the order due to missing ingredients"
    }).promise().then(data => console.log(data));
}
}

sfn.getActivityTask({
    activityArn: "arn:aws:states:us-west-
        2:495525968791:activity:prep_order_activity"
}).promise().then(data => {
    processData(data);
});
```

In this snippet, we first get the activity and pass it to the method `processData`. This code snippet doesn't show any implementation for the preparation because it can be anything related to the requirement or waiting for human interaction. This process sets a success or failure flag before informing SFn. The `order` object is passed on as input to the task. If the process is successful, the `status` property of the `order` object is set to PREPPED.

If the preparation task doesn't yield a positive result, we invoke the `sendTaskFailure` method and pass `error` and `cause` along with `taskToken`. `Error` is meant for a short code, whereas the attribute `cause` is to give more detailed information about what caused the error. This information will be recorded in the state machine execution. The sample executions with both results are shown in Figure 5-13.

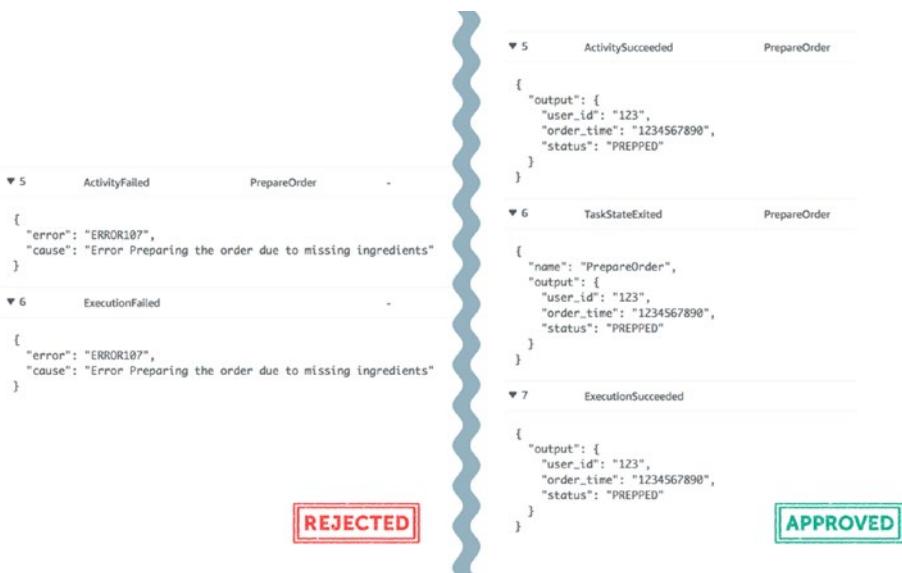


Figure 5-13. *SendTaskSuccess* vs. *SendTaskFailure*

As you can see in this figure, in case of failure, the given error and cause are sent to the given execution. If the state machine has retries attached to this state, it will be retried; otherwise, the same information is shown to be the cause of the execution failure. In the success case, the output with an additional field of status appended to the given order object is passed on to the next states as input.

As you have known already, the state also has a heartbeat timeout. Our worker should also keep in mind the regular updates it has to send to the state machine to let execution wait until the task completes. It is especially important if the task is about a human interacting with the tablet he is operating to perform the task.

You might have seen similar examples earlier when you go to an ATM. If you don't finish withdrawal of the money within the time limit, it asks you if you need any more time to finish the transaction. If you say Yes, the ATM will wait for some more time. So the guy who is cooking the order may need more time than previously thought, so he will have to press that he is still working on that same order so that SFn will keep the task in running state.

If the chef responds with Yes when asked with the question "Do you need more time?" the worker sends a heartbeat signal to the SFn with the given task token. It is an indication that the task is still being worked on and shouldn't be timed out. The following snippet shows how to send the heartbeat:

```
sfn.sendTaskHeartbeat({  
    taskToken: data.taskToken  
});
```

This mechanism is especially useful when the worker crashes for whatever reason. In such cases, SFn doesn't receive any heartbeats so it can time out the state and retry if configured. You can take any number of approaches to decide when you send the heartbeats regularly. The interval between each heartbeat signal can be a fixed amount of time in a background thread. Alternatively, we can also send a heartbeat after each subtask is over.

Other Patterns

In this section, let us see some example problems we encounter on a regular basis in our production systems. We may have to invoke a step function from another step function to modularize the step function definitions if they are too large to fit in one. We may need to run long-running batch jobs. Let us see how to solve these commonly occurring problems.

Long-Running Tasks

In the previous section, we have seen one mechanism to control when the SFn considers a state to have timed out. To keep the state in running mode, we can regularly send heartbeats to send a signal to prevent the task from timing out. This is an example of a request-response mechanism.

However, there are few tasks which are not in our control. We may have to sometimes rely on services that cannot send a signal on a regular basis. For example, AWS Batch/Glue jobs may take any amount of time depending on the size of the tables being queried or the joins being used. We may wait for a message to arrive at an SQS queue or a Lambda to finish executing.

These services can't send a heartbeat, like our activity tasks. SFn provides two mechanisms named syncing and waiting for task token to solve the problems. Some services support wait-for-task-token, and some support sync operation. Here is the complete list of integrations available as per the AWS documentation. Table 5-5 has been directly lifted out of that documentation.

Table 5-5. Step Functions integration patterns with other AWS services

| Service | Request/Response | Sync | Wait-for-task-token |
|------------------|------------------|------|---------------------|
| AWS Lambda | ✓ | | ✓ |
| AWS Batch | ✓ | ✓ | |
| DynamoDB | ✓ | | |
| ECS/Fargate | ✓ | ✓ | ✓ |
| SNS | ✓ | | ✓ |
| SQS | ✓ | | ✓ |
| AWS Glue | ✓ | ✓ | |
| Amazon SageMaker | ✓ | ✓ | |
| SFn | ✓ | ✓ | ✓ |

As you can see from the table, all the services support the request/response pattern of communication for short-running tasks. However, for long-running tasks, most of them only support sync or wait for task token mechanism. As we can build services of any level of complexity using ECS/Fargate with complete control, those services allow all kinds of patterns.

Sync

Sync is a pattern used to run the jobs in the integrated AWS services typically. When SFn asks AWS to run a job, it needs to wait for the job to complete. To ask SFn to wait for the job to be completed, we need to specify the keyword sync at the end of the resource ARN as shown in the following snippets:

```

"Start Glue Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::glue:startJobRun.sync",
    "Parameters": {
        "JobName": "orderBackupJob"
    },
    "Next": "AnalyticsState"
}

"DeliveryState": {
    "Type": "Task",
    "Resource": "arn:aws:states:::ecs:runTask.sync",
    "Parameters": {
        "LaunchType": "FARGATE",
        "Cluster": "arn:aws:ecs:REGION:ACCOUNT_ID:cluster/
MyECSCluster",
        "TaskDefinition": "arn:aws:ecs:REGION:ACCOUNT_ID:task-
definition/MyTaskDefinition:1"
    },
    "End": "true"
}

```

Here is an example of two different states. The first one is a Glue job task. It is a task that triggers the job named orderBackupJob and waits until the job is finished. Glue jobs are essentially ETL jobs. Initiating the job is a synchronous process. This integration is a great fit for workflows that need to process large amounts of data.

The second example state triggers an ECS task and waits until the task is finished. We use this integration when we want to launch containers that run in EC2 instances or Fargate. The waiting part is optional and is because of the sync keyword at the end of the resource ARN. When the respective tasks are completed, these services send a signal to SFn to continue to the next states.

Task Token Callbacks

Earlier, I have listed SQS as a service that supports wait-for-task-token pattern of integration. ECS also supports this mechanism. Third-party systems can utilize task token mechanism for tasks that can take an indefinite amount of time. In this pattern, SFn waits for a specific task token to be received from the worker.

Imagine a scenario where a step function sends a message to an SQS queue and it has to wait until that message is consumed by another consumer that is not invoked by this state machine. In such cases, how does SFn know that the consumer finished its task? The state machine may have no knowledge of the consumers of the queue and doesn't get notified when the message has been consumed or is being processed.

In such cases, wait-for-task-token pattern is useful. When a message is sent to SQS, the message also includes a task token. This token is then passed on to the consumer of the queue as usual, and the consumer has to respond to Step Functions using sendTaskSuccess API. The flow works as shown in Figure 5-14.



Figure 5-14. Wait for Task Token Pattern

In this approach, SFn doesn't have to concern itself with invoking lambda function, yet still is able to wait until Lambda does its job. We can decide the message body structure in the state machine definition. The task token can be retrieved from the context using `$$.Task.Token`:

```
"Start Task And Wait For Callback": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
        "TopicName": "TaskQueue",
    "Parameters": {
        "Message": {
            "MessageTitle": "Task started by SFn. Waiting for
                callback with task token.",
            "MessageBody": {
                "TaskToken.$": "$$.Task.Token"
            }
        },
        "Next": "Notify Success",
        "Catch": [
            {
                "ErrorEquals": [ "States.ALL" ],
                "Next": "Notify Failure"
            }
        ]
    }
}
```

This message body has to be parsed in the lambda function or the worker consuming the message using regular mechanisms. A typical Lambda function handler written in NodeJS is shown here. In here, we use

CHAPTER 5 STEP FUNCTIONS

the JSON.parse method to convert string to a javascript object and read the task token. We create params object where we need to have a parameter named taskToken with the value received through message body while invoking sendTaskSuccess API:

```
const aws = require('aws-sdk');

exports.lambda_handler = (event, context, callback) => {
    const stepfunctions = new aws.StepFunctions();

    for (const record of event.Records) {
        const messageBody = JSON.parse(record.body);
        const taskToken = messageBody.TaskToken;

        const params = {
            output: "\"Callback task completed successfully.\"",
            taskToken: taskToken
        };

        console.log(`Calling SFn to complete task with params
        ${JSON.stringify(params)}`);

        stepfunctions.sendTaskSuccess(params)
            .promise()
            .then(data => console.log("Successfully sent the
            tasktoken back"));
    }
};
```

In this snippet, we send the taskToken back as is using the sendTaskSuccess API. In case of issues while processing the input, We can invoke sendTaskFailure API to allow SFn to retry the task if the state machine definition has it configured for the state that sends the message to SQS. If the sendTaskSuccess or sendTaskFailure is not invoked, usual timeout configurations of the state machine apply.

Child Step Functions

All the step function examples we have until now invoke other AWS services. SFn also allows you to invoke a step function from another step function's state. The state is of type Task. To understand how to invoke a state machine, keep a note of the data that is necessary for running a state machine in the usual way.

As we need to invoke a specific state machine, we definitely need to have the ARN. While executing a state machine, we need to provide an input for the state machine to process. So to invoke another step function as a child workflow, we need to provide the ARN and input, as shown in this state machine snippet. The information has to be passed in while invoking state machine through the Parameters attribute:

```
"Start new workflow and continue asynchronously": {  
    "Type": "Task",  
    "Resource": "arn:aws:states:::states:startExecution",  
    "Parameters": {  
        "StateMachineArn": "arn:aws:states:us-west-2:495525968791:  
                           stateMachine:ChildStepFunction",  
        "Input": {  
            "Param1": "value1",  
            "PARENT_EXECUTION_ID.$": "$$.Execution.Id"  
        }  
    },  
    "Next": "NEXT_STATE"  
}
```

The preceding snippet starts execution of another step function and continues without waiting for the execution to finish. If you want to wait for

the execution to finish, you can use sync/waitForTaskToken keyword at the end of the resource attribute as shown here:

```
arn:aws:states:::states:startExecution.sync arn:aws:states:::states:startExecution.waitForTaskToken
```

It is not necessary to pass parent execution id to the child step function. You can pass parent execution id as shown in the preceding text if you want to log it for auditing purposes.

Creating Loops

Let us now move on to another practical scenario. We have to often repeat a state for an arbitrary number of times during a state machine execution. It may not be possible to know that number beforehand. Let us think of another functionality in our order restaurant.

Even though we don't have the signup feature in our application, assume that for all the customers who sign up in a day, we have to copy their profile pictures and resize them to optimize them for the mobile app. This resize operation on all the profile pictures can be a new state machine. The state that performs the optimize operation needs to be repeated, and number of users who sign up in a day are totally variable.

There are no new functionalities in SFn to be explored; we can do this with the features we are already aware of. We can use a Choice state combined with other state types to loop. Here is a simple pseudo-code format of the state machine definition that achieves looping:

1. State to Seed List of New Customers, Move to State 2
if seeding is successful.
2. Choice state to decide if more customers are available.
 - a. If no more customers, move to State 4.
 - b. If there are customers, move to State 3.

3. Process the customer photo and reduce the count of customers and move to State 2.
4. Final state to finish the execution.

Each state has to pass the necessary information to enable the decision-making process in the Choice state. One approach is that the seeder lambda function can add a new property named count and set it to be equal to the size of the list. An example of a seeder state output and choice state input can be as shown in the following:

```
{
  "userProfilePictures" : [
    "s3://awsome-restaurant/profile-pics/user001.png",
    "s3://awsome-restaurant/profile-pics/user002.png",
    "s3://awsome-restaurant/profile-pics/user003.png",
    "s3://awsome-restaurant/profile-pics/user004.png",
  ],
  "count" : 4
}
```

I'm not going to show the implementation of the seeder lambda as it is immaterial to how the state machine works. The crux of this section is how we implement the logic of the Choice state. Here is one way to define the state:

```
"loop": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.count",
      "NumericGreaterThan": 0,
      "Next": "ProcessImage"
    }
  ]
}
```

CHAPTER 5 STEP FUNCTIONS

```
],
  "Default": "FinalState"
}
```

This state transitions the machine to ProcessImage state if the count is more than zero. Otherwise, the machine will be transitioned to FinalState. The machine must transition to loop state every time the ProcessImage task is completed:

```
"ProcessImage": {
  "Type": "Task",
  // rest of the state definition
  "Next": "loop"
}
```

This state invokes a lambda function named img-optimizer and waits until the execution is completed. Once the Lambda function completes processing the image, the machine will be transitioned to loop choice state. The lambda function may process one or more images and removes the images from the list and reduces the count by the number of processed images. Assuming that the image optimizer processes first two images specified in the list of images, the output of this state will be as follows:

```
{
  "userProfilePictures" : [
    "s3://awsome-restaurant/profile-pics/user003.png",
    "s3://awsome-restaurant/profile-pics/user004.png",
  ],
  "count" : 2
}
```

The loop will trigger the process image state as the count is more than zero, and the same process continues. Once the count reaches the value zero, the state machine moves to FinalState, which will end the execution

of the state machine. Putting all of this together, this is how the state machine visualization, as shown in Figure 5-15, looks like.

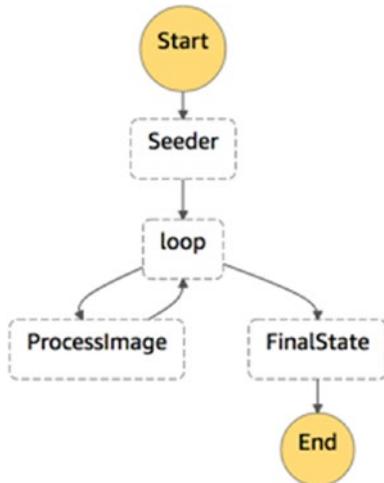


Figure 5-15. Visualization of the Loop

```
{  
  "StartAt": "Seeder",  
  "States": {  
    "Seeder": {  
      "Type": "Task",  
      "Resource": "...lambda:invoke.waitForTaskToken",  
      "Parameters": {  
        "FunctionName": ".....function:seeder:$LATEST",  
        "Payload": {  
          "Input.$": "$",  
          "TaskToken.$": "$$.Task.Token"  
        }  
      },  
      "Next": "loop"  
    },  
  },
```

CHAPTER 5 STEP FUNCTIONS

```
"loop": {
    "Type": "Choice",
    "Choices": [
        {
            "Variable": "$.count",
            "NumericGreaterThan": 0,
            "Next": "ProcessImage"
        }
    ],
    "Default": "FinalState"
},
"ProcessImage": {
    "Type": "Task",
    "Resource": "...::::lambda:invoke.waitForTaskToken",
    "Parameters": {
        "FunctionName": "...::::function:img-processor:$LATEST",
        "Payload": {
            "Input.$": "$",
            "TaskToken.$": "$$.Task.Token"
        }
    },
    "Next": "loop"
},
"FinalState" : {
    "Type": "Pass",
    "End": true
}
}
```

I have truncated resource and function names to keep the code simple. Both the tasks use waitForTaskToken pattern to keep the tasks synchronous. The number of iterations is totally controlled by how

many records are found by the seeder. We have covered almost all the functionalities provided by the state definition language. Let us move on to other topics.

Monitoring

In any production system, it is essential to keep track of the running state machines, how they are progressing, how much time they are taking to execute, and so on. We can use CloudWatch to monitor and collect metrics related to state functions running in our AWS accounts. We can look at the metrics related to Step Functions by opening CloudWatch and navigating to the following path, CloudWatch ➤ Metrics ➤ All Metrics ➤ States, as shown in Figure 5-16.

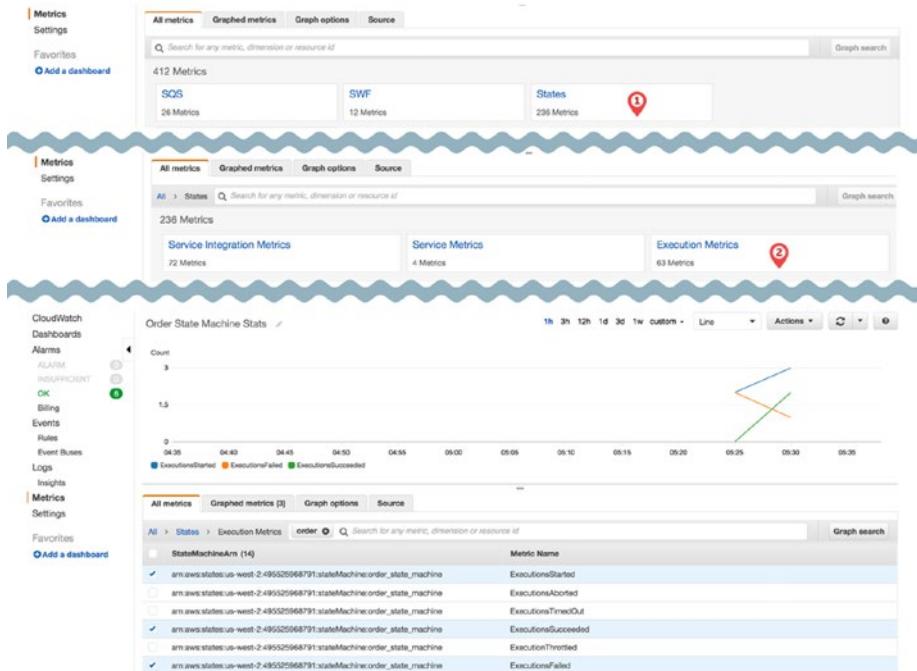


Figure 5-16. Execution Metrics of Step Functions

CHAPTER 5 STEP FUNCTIONS

There are different metrics captured automatically by CloudWatch for monitoring without any more work from our end. All we have to do is choose which metrics we want to graph and what dimensions we want to use. The preceding graph is plotted from three different metrics, execution started, executions succeeded, and executions failed, for the state machine named order_state_machine. You can search with the name of the state machine if there are many. The preceding screenshot shows search results of keyword order inside execution metrics.

The preceding graphs show the number of executions in every 5 minutes block for the last one hour. However, if you select the same metrics in your CloudWatch console, you may see graphs with different values. It is because by default, the statistic is average. You can go to the Graphed metrics tab to choose a different statistic and time period. The options I have chosen are as shown in Figure 5-17.



Figure 5-17. Graph Options to Change Statistic and Period

There are a few other metrics we can choose from. The most useful metric we may need to keep an eye on is execution time. Using this metric, we can keep track of how much time it is taking to complete each execution. In our restaurant, we may want to aim to complete whole PREP, COOK, PACK, and SHIP states within less than 1 hour.

Another important statistic we are interested in is any execution failures. Ideally, we should not have any execution failures. We can also graph the metrics associated with other integrations such as Lambda or DynamoDB throughput to keep track of other issues that are not directly related to step function but can eventually cause issues with state machine executions.

To ensure that our customers are satisfied, we need to look out for instances that cross these thresholds constantly. So we can create an alarm on these metric graphs. Creating alarms is very simple. All we have to do is click the bell icon beside the metric we have chosen inside the Graphed metrics tab, as shown in Figure 5-17.

Alternatively, we can also create an alarm by going to the Alarms view in the CloudWatch metrics left-hand menu bar and clicking the Create Alarm button. This opens up a new wizard where you can select a metric on which you want to create an alarm.

In either case, we are presented with several options to create an alarm. We need to choose a statistic and a time period to begin with. We also have to select a condition and the threshold. The graph reflects the alarm threshold and the current state of statistic in relation to the threshold. This has been represented in Figure 5-18.

The current threshold in the figure is 350, which is represented with a red horizontal line in the graph. The blue line trying to reach the threshold is the maximum execution time in the last 5 minutes. Whenever it does, an alarm is raised, and the action configured in the next screen will be executed. The available actions in the next page are sending a notification to SNS or auto-scaling EC2 instances. If we need more actions, we can always perform them through a Lambda subscribed to the SNS topic to which notification is sent.

There are many configurations possible including creating different types of graphs such as stacked area and number. We can also configure anomaly detections or add other annotations to quickly recognize the issues that are going to come even before the threshold is breached. We can create a horizontal annotation at 80% of the threshold at which the alarm is raised so that a user looking at the graph can notice the issues before they arise.

CHAPTER 5 STEP FUNCTIONS

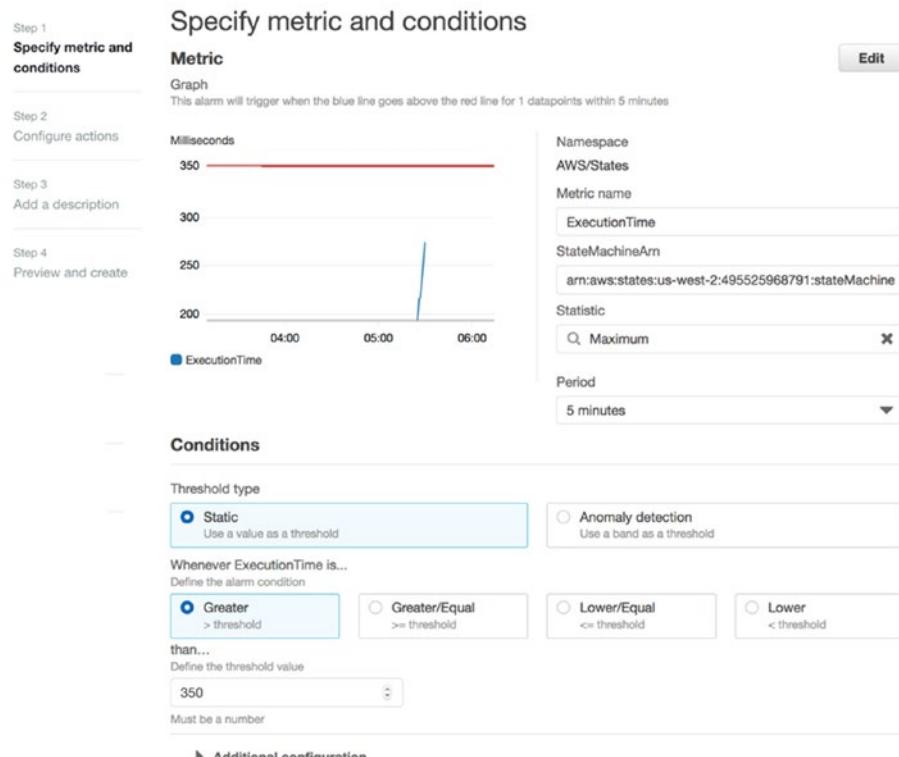


Figure 5-18. Alarm Options to Configure a Threshold

We can also use AWS CloudTrail to record actions taken by users on roles or step functions to keep a record of the events performed. SFn supports logging of events such as CreateActivity, CreateStateMachine, StartExecution, and UpdateStateMachine in CloudTrail. Event entries are created whenever these operations are performed on state machines.

Opening CloudTrail will show a list of events as shown in Figure 5-19. You can access CloudTrail just like any other service from the AWS console. Once in CloudTrail, you can go to the Event history page by clicking the corresponding link in the left-side navigation menu.

The screenshot shows the CloudTrail Event history interface. On the left, there's a sidebar with links like CloudTrail, Dashboard, Event history (which is selected), Trails, Learn more, Pricing, Documentation, Forums, and FAQs. The main area has a title 'Event history' and a sub-instruction: 'Your event history contains the activities taken by people, groups, or AWS services in your account. By default, the view filters out read-only events. You can change or remove that filter, or apply other filters.' Below that is a note: 'You can view the last 90 days of events. Choose an event to view more information about it, if your CloudTrail events, create a trail and then go to your Amazon S3 bucket or CloudWatch Logs. Learn more'.

There's also a search bar with placeholder text 'Can't find what you're looking for? Run advanced queries in Amazon Athena' and a toolbar with icons for refresh, download, and settings.

The central part is a table with columns: Event time, User name, Event name, Resource name, and Error. Two rows of data are shown:

| Event time | User name | Event name | Resource name | Error |
|-------------------------|-----------|----------------|---------------|-------|
| 2019-08-25, 10:33:26 PM | admin | StartExecution | | |
| 2019-08-25, 10:33:07 PM | admin | StartExecution | | |

When the second row is expanded, it reveals detailed event information:

| AWS access key | Event time | AWS region | Error code | Event ID | Source IP address | User name |
|-----------------------|-------------------------|------------|------------|--------------------------------------|-------------------|-----------|
| ASIAJXG3SGHOL7NM83DUE | 2019-08-25, 10:33:07 PM | us-west-2 | | 736a4a7d-8e3f-44bb-9355-f8298015e269 | 71.227.158.154 | admin |
| | | | | Event name: StartExecution | | |
| | | | | Event source: states.amazonaws.com | | |

Below this, a section titled 'Resources Referenced (0)' is shown. At the bottom of the expanded row, there's a 'View event' button.

Finally, another row of data is shown below the expanded row:

| Event time | User name | Event name | Resource name |
|-------------------------|-----------|--------------------|---------------|
| 2019-08-25, 10:33:04 PM | admin | UpdateStateMachine | |

Figure 5-19. CloudTrail Event History of Step Function Events

In the figure, you can see that we have filtered out events coming from states. Even though they are created by SFn, it is often referred to as states in internal documentation. By expanding each event in the table, you can see more information such as which access key has been used and what is the source IP address.

If you want to see more details, you can click the View event button inside each event. This will bring up a modal window with a lot more details in it. CloudTrail automatically records events from all the services; however, they will be removed automatically after 90 days. You can configure CloudTrail to retain the records in an S3 bucket and records setting up metrics and alarms; however, they are not in the scope of this book.

Updating Order Workflow

We have covered much ground so far. Not just this chapter, we have been learning so much stuff leading to this final chapter to make sure that we create an orchestration workflow that works smoothly to ensure our

CHAPTER 5 STEP FUNCTIONS

application runs with little friction. So, in this section, we will put together many concepts we learned until now to come up with a complete state machine.

We have discussed several state types, one of them being a Task. Majority of the work is being carried out by a Task state as explained before. To convert our order workflow to a state machine, let us look at each of the steps again once we receive a request for a new order. Earlier, our Lambda functions were doing the bulk of the work like inserting the record in DynamoDB, starting workflows created using SWF, inserting data in queues, and so on, as Step Functions by default eliminates the need to do the heavy lifting.

SFn can insert the data in DynamoDB or update it as well, so lambdas don't have to concern themselves with that part. This way, the state of the execution is entirely maintained by the state machine. As SFn can wait for a lambda to be available to get started and pick up a task, there is no need to buffer the input queue with an SQS queue. This service acts like a queue or a notifier because it can send messages to SNS:

1. Insert the order record in DynamoDB with order CONFIRMED status.
2. The PREP stage can execute both of these either in parallel or sequentially.
 - a. Once the order is confirmed, we need to send the message to the customer that the order is confirmed.
 - b. Once the data is inserted, we need to move the order to the PREPARE state, which is a lambda function that performs the work of changing the order status to PREPARED.

3. If this is successful, it should then move to COOK state, which is another lambda function to perform the work. Once cook lambda performs its job, it changes the status to COOKED.
 - a. Once cooking is completed, we need to send a notification to the customer.
4. If the COOK stage is completed successfully, we need to perform two activities. Move the order to the PACKAGE stage and simultaneously notify the delivery agent to get ready to pick the order.
 - a. Once packaging is completed, we should update DynamoDB item's status to PACKED.
5. We need to inform the delivery agent to pick the order and notify the customer that the order is on the way.
6. Once the shipping operation is completed, we need to mark the order as COMPLETE.

If any of these states result in an error, we can configure the retry up to three times without any backoff. We need to retry the activity as soon as possible because the order has to be delivered to the customer on time. So, ideally, we shouldn't be increasing the time gap between each retry.

The following state takes care of inserting the data in DynamoDB. This expects that the input for the state function contains the fields user_id, order_time, food_id, and total_amount:

```
"InsertDynamoDBEntry": {  
    "Type": "Task",  
    "Resource": "arn:aws:states:::dynamodb:putItem",  
    "Parameters": {  
        "TableName": "orders",
```

```

    "Item": {
        "user_id": {
            "S.$": "$.user_id"
        },
        "order_time": {
            "S.$": "$.order_time"
        },
        "food_id": {
            "S.$": "$.food_id"
        },
        "order_status": {
            "S": "CONFIRMED"
        },
        "total_amount": {
            "S.$": "$.total_amount"
        }
    }
}

```

The next four stages follow a similar pattern of invoking either a lambda function or ECS task or such and then perform an update operation in DynamoDB so that other applications asking for the same order information may have the latest status:

```

"PrepareOrder": {
    "Type": "Task",
    "Resource": "arn:aws:states:::ecs:runTask.waitForTaskToken",
    "Parameters": {
        "LaunchType": "FARGATE",
        "Cluster": "arn:aws:ecs:::cluster/OrderCluster",
        "TaskDefinition": "arn:aws:ecs:::task-definition/
                        PrepareOrder:1",
    }
}

```

```
"Overrides": {
    "ContainerOverrides": [
        {
            "Name": "CLUSTER_NAME",
            "Environment": [
                {
                    "Name": "TASK_TOKEN_ENV_VARIABLE",
                    "Value.$": "$$.Task.Token"
                }
            ]
        }
    ],
    "Next": "SetOrderToPreppedState",
    "Retry": [
        {
            "ErrorEquals": [ "States.All" ],
            "IntervalSeconds": 1,
            "BackoffRate": 1,
            "MaxAttempts": 3
        }
    ]
},
"Update the order status to Prepped": {
    "Type": "Task",
    "Resource": "arn:aws:states:::dynamodb:updateItem",
    "Parameters": {
        "TableName": "orders",
        "Key": {
```

```
        "user_id": {
            "S": "$.user_id"
        },
        "UpdateExpression": "SET order_status = :myValueRef",
        "ExpressionAttributeValues": {
            ":myValueRef": {
                "S": "PREPPED"
            }
        }
    },
    "Next": "COOK_STATE"
}
```

I'm not going to present the rest of the states as they follow a similar approach to invoke a lambda function or ECS task and then invoke a DynamoDB update. I want you to give it a shot in creating the implementations of either a lambda function that performs these tasks or ECS tasks. I have not covered creating ECS tasks in this book so as not to complicate or cover topics that don't need to be learned absolutely. Make sure you try to learn how to integrate ECS with SFn.

Step Functions vs. SWF

As you might have noticed, we are essentially doing the same things. We have built a workflow to keep an entire execution together in both Chapters 4 and 5. However, each of these two services approaches the problem in a different manner; and if you have tried to build your

applications, you would appreciate how easy it is to get started with Step Functions when compared with SWF. Step Functions has the following advantages:

- Completely serverless – SWF needs at least deciders to run on a persistent compute such as EC2 or such. Step Functions doesn't have this requirement.
- A developer can create the complete workflow from the AWS console as opposed to SWF where the user defines few tasks in the console and some orchestration.
- Step Functions is way cheaper to run because you are not charged for the time SFn is waiting for tasks to be completed. Instead, you are charged based on the number of state transitions. In SWF, a decider has to regularly poll while the tasks are being run.
- Step Function creation can be entirely done automatically using CloudFormation or CDK; however, only infrastructure-related entities can be automatically managed.
- Easy to debug and find the root cause of the failure in the case of Step Functions. We don't even have to look into logs to figure out where a step function has failed or got stuck. Just looking at the visual representation of the execution can sometimes give clue as to what is wrong.

With all these advantages, Step Functions is the clear winner when it comes to picking a service at this stage, with a lot more new features in recent past when compared with SWF. The might of AWS is behind SFn. Even the SWF FAQs page contains the following blurb that clearly suggests that you should go with SFn over SWF:

Q: When should I use Amazon SWF vs. AWS Step Functions?

AWS Step Functions is a fully managed service that makes it easy to coordinate the components of distributed applications and microservices using visual workflows. Instead of writing a Decider program, you define state machines in JSON. AWS customers should consider using Step Functions for new applications. If Step Functions does not fit your needs, then you should consider Amazon Simple Workflow (SWF). Amazon SWF provides you complete control over your orchestration logic, but increases the complexity of developing applications. You may write decider programs in the programming language of your choice, or you may use the Flow framework to use programming constructs that structure asynchronous interactions for you.

As the statement from AWS indicates, it is best if you put your bets on Step Functions to have better support and better knowledge base and maximum support from the AWS team as it may eventually maintain two different systems, and as of now Step Functions is mostly likely to get the bigger pie.

Pricing

Step Functions's pricing is one of the simplest strategies. You are charged for the number of transitions you make including retries. Assume you have a HelloWorld state machine that just has a beginning state, a process state, and an end state. So it will transition from beginning state to process state

and then to end state. So there are two transitions. Each transition costs \$0.000025. If you run this simple pipeline for 100000 times, the cost is as in the following:

$$2 * 100000 * \$0.000025 = \$5$$

The first 4000 transitions are deducted as free tier per month even after the initial 12 months of the trial period. So the cost of the preceding pipeline is actually \$4.9. As always, these are current rates, and they can change all the time. However, there are other costs you are going to incur such as data transfer costs, and if you are running lambda functions for tasks, their compute also costs separately. So the true cost of a workflow is a combination of all these individual elements.

Summary

In this chapter, we have seen how to use Step Functions to create optimized workflows without the need to create always up and running deciders and activity workers. We have begun the chapter by learning the differences between SWF and Step Functions and understanding what SFn brings to the table where there is already a service for creating workflows.

We have then seen what a state machine is and understood the key concepts necessary for us to understand this service effectively. We have understood what states, activities, and tasks in the SFn world are. We have then started by creating a simple hello world state machine before we go into the details of the state definition language.

The state definition language allows us to create state machines for solving real-world problems of orchestration with ease. We have seen how to create various types of states. The most versatile type of state is of type Task. We have seen how to create Task states and implement workers to do the work. We have also seen examples of how to create Pass, Wait, Success, and Fail states apart from the Task state.

We have then seen how to use Choice and Parallel states to create decision and parallel branches. The Choice state allows us to transition the state to different branches based on the conditions we define. The Parallel type of state allows us to run multiple states simultaneously.

We have also gone over how to handle errors by making use of retry and catch clauses of state definitions. These clauses allow us to specify the number of retries and backoff rate in case of consecutive failures.

SFn also allows you to specify various timeouts like heartbeat and task timeouts along with execution timeout for the entire state machine execution. These timeouts allow us to make sure that our state machine executions are not forever stuck executing because of a crashed server or faulty input.

We have then seen how to create activities and activity workers so that we can create non-AWS worker. We need to tell SFn about the activities that are going to be performed outside of the AWS cloud and provide implementations. Activity workers need to get tasks waiting to be executed using GetTaskActivity API and perform the necessary work.

We have also seen numerous other patterns such as sync and waitForTaskToken to run long-running tasks. Along with it, this chapter also demonstrated how to create loops using the Choice state and invoking child step functions. This helps us modularize state definitions of complex step functions.

Finally, we have also seen how to monitor the SFn activities and executions to keep track of any errors. We can also create alarms using these metrics accessed through the CloudWatch service of AWS. Before we end the chapter, I have also demonstrated how to plan for implementing our order workflow using the state definition language.

Index

A, B

Amazon Machine Image (AMI), 113, 118
Amazon Web Services (AWS), 1
account, creation, 6
compute section, 3
cost management, 5
database section, 4
storage section, 4
API method, 273–276
Application, restaurant
home view, 50, 51
lambda function, 51
package.json file, 53
project structure and organization, 52
S3, 52
Vue component, 53
example, 54, 55
FoodItem DOM element, 56
format, 54
HTML-based template, 54
mutation methods, 56
on_increment_click, 55, 56
VueJS, 49
AWS flow framework, 256–257

aws help command, 17
AWS infrastructure, 191
AWS Management Console, 2, 3
AWS SAM, 44
AWS Toolkit plugins, 44

C

Child step functions, 323, 324
CloudWatch events, 237, 242
Cognito, 25, 32, 58
Consumption approaches
EC2 NodeJS SQS worker, 112, 113
aws-sdk package, 119
install, 118
instances, 116, 117
launching, 113–115
pm2 logs, 125, 126
process manager, 124, 125
rule, SSH, 115
sqs-consumer, 120, 121, 123, 124
ssh-ec2.pem, 116
lambda SQS worker, 98, 99
creation, 99–102
event to mimic, 105, 106
hard-coding, 110

INDEX

- Consumption approaches (*cont.*)
invoking, 102–105
orchestration, 107–110, 112
poller blueprint, 100
create-queue command, 70, 72
createStateMachine method, 273
- D**
- Dead letter queue (DLQ), 133
Decider, implementation
 decision making
 events list, 230
 handleOrder method, 222
 infinite loop, 230
 nextPageToken property, 232, 233
 orchestrate method, 223
 polling cycle, 227–229
 query method, 223
 ScheduleActivityTask, 224
 source code, 226, 231, 232
 SQS consumer, creation, 230
 types, 224, 225
 decision tasks, 217–222
deleteStateMachine method, 276
DynamoDB, 5
 AWS SDK, 22, 24
 find data, 21, 22
 NoSQL database service, 18
 RDBMS, 19
 SDK, insert data
 client, create, 27
 Cognito, 25
- IAM roles, 25
identity pool, 25
mvn install command, 30
program structure, 24
putItem method, 31
table, 28
SDK, query data, 32–34
tables and insert items
 Create Item wizard, 20
 hash key, 19
 read and write capacity units, 19
- DynamoDB from Lambda
 amount of memory, 40
 Execution role, 39
 IAM role, 39
 insert data, table, 40, 41
 Java, 42, 43
 putItem method, 41
- DynamoDB table/lambda function, 70
- E**
- Eclipse, 44, 256
Elastic Beanstalk, 49
Elastic Container Service (ECS), 49, 265, 319
- F, G**
- Firebase Cloud
 Messaging (FCM), 173
First-in-first-out (FIFO) queues, 66

creation, 127, 128
 DLQ, 132–134, 136, 137
 receive messages, 130, 131
 receive messages, request
 fails, 132
 sending messages, 128, 130

H

handleMessage function, 120

I

Identity and access
 management (IAM), 57
 admin user, 8, 10
 AWS Management Console, 7
 MFA, 8
 IntelliJ, 44

J, K

JSON.stringify() method, 135

L

Lambda
 access, DynamoDB (*see*
 DynamoDB from Lambda)
 CloudWatch, logs, 35, 47–49
 function, creation
 CloudWatch service, 37
 configuration, 36, 37

Create Function page, 36
 custom runtime, 36
 exports.handler, 37
 Test button, 38
 from HTML page
 AWSLambdaInvocation-
 DynamoDB, 47
 Cognito, 45, 47
 mechanisms, 44
 pricing model, 35
 Launch Instance button, 113
 LightSail, 5
 Long polling, 92–93

M

Monitoring, step functions
 CloudTrail Event, 333
 execution metrics, 329
 metric graphs, 331
 statistic and period, 330
 threshold configuration, 331, 332
 Multifactor
 authentication (MFA), 8

N

Node version manager (NVM), 118
 npm install command, 23, 53

O

Orchestration rules, 107, 109

INDEX

P

processOrders function, 109
putItem method, 29

Q

QuickSight, 4

R

Receive Count column, 85
Redrive Policy, 68, 134
Register activity type
 API method, 206, 207
 CLI method, 205
 console method, 204, 205
Register domain
 API method, 198
 CLI method, 195–197
 console method, 193, 195
Register workflow type
 API method, 202, 203
 CLI method, 201, 202
 console method, 199–201

S

save_order method, 157
ScheduleLambdaFunction, 238
sendMessage method, 82
Serverless workflow
 activity workers, 238–241
 cancel timer, 251–253
 deciders
CloudWatch events, 242, 244, 247
Cron expression, 245
lambda function, 242, 243, 246
rate expression, 245
role and timeout configuration, 243
SWF functionalities, 243
frameworks, 256, 257
heartbeats recording, 253–255
limits, 259, 260
pricing, 257–259
start timer, 248–250
setOrderStatus method, 121
setQueueAttributes operation, 135
Short polling, 92
Simple notification service (SNS)
 access policy, 185, 187
 configuration, 184
 message filtering
 add filter policy, 165, 166
 attribute, 167
 blacklisting, 169
 conditions, 164
 existence check, 171
 multiple conditions, 172
 number range
 matching, 170, 171
 prefix matching, 169
 whitelisted value, 168
 message receiver
 lambda receiver, 158–160
 SQS, 162

publish message
 API method, 153, 155–158
 JSON map, 152
 open wizard, 150, 151

push notifications, FCM
 client endpoint, 182–184
 enable receiving
 notifications, 173, 174
 firebase steps, 174, 175
 initializing in web-app,
 176–178
 SNS setup, 179–181

setting up
 create subscription, 146–150
 topic creation, 142–145

SQS, 140

Simple queue service (SQS), 140
 architecture, 62
 chef service, 63
 component/application, 60
 component interaction, 61
 consumption (*see* Consumption
 approaches)
 creation (*see* SQS queue
 creation)
 decoupled services, 65
 delivery process, 64
 multiple components, 62
 SMTP server, 61
 types, 63
 waiter service, 63

Simple Storage Service (S3)
 Amazon.com, 10
 CLI, 15–17

static web site
 bucket tab, 12, 13
 Create Bucket
 button, 12
 index.html, 11
 NoSuchWebsite
 Configuration, 13, 14
 URL, access, 15

Simple Workflow Service (SWS)
 architecture, 190–193
 implementation
 activity worker, 233–237
 decider, 217
 starter, 212

register activity type
 API method, 206, 207
 CLI method, 205
 console method, 204, 205

register domain
 API method, 198
 CLI method, 195–197
 console method, 193, 195

register workflow type
 API method, 202, 203
 CLI method, 201, 202
 console method, 199–201

SNS publish operation, 290

SQS queue creation
 CLI command, 70–73
 configuration wizard, 69
 console, 66, 68
 SDK method, 74–77

SQS queue message, life
 cycle, 93–96

INDEX

SQS queue pull message
 CLI method, 86–88
 console, 84, 86
 long polling, 92, 93
 SDK method, 84, 88, 90, 91

SQS queue send message
 CLI method, 80–82
 console, 78, 79
 delayed, 79
 optional attributes, 80
 SDK method, 82, 84

Starter, implementation
 childPolicy, 213
 deployment process, 216
 domain, 212
 executionStartToClose
 Timeout, 213
 input, 213
 lambda handler code, 214, 215
 lambdaRole, 213
 SimpleWorkflowFullAccess, 216
 tagList, 213
 taskList, 212
 taskPriority, 212
 taskStartToCloseTimeout, 213
 workflow execution code, 213, 214
 workflowId, 212
 workflowType, 212

State definition language (SDL), 269

State machine
 choice state
 conditional operators, 303–306
 fields, 299
 operators, 300, 301
 rules, 300
 visual rendition, 302

creation, 278

fail state, 298

fields, 280

parallel state, 306–308

pass state, 281
 parameters, 284–286
 result, 282
 ResultPath, 283–284

succeed state, 298

task state
 catch field, 294–296
 parameters, 291, 292
 resource, 290
 ResultPath, 297, 298
 retrier objects, 292–294
 TimeoutSeconds, 296, 297

top-level fields, 277, 278

wait state, 287, 288

Step functions (SFn)
 activity, 265
 activity worker
 creation, 309–311
 implementation, 311–316

API method, 273–276

long-running tasks, 317, 318

loops creation, 324–329

pricing, 340, 341

state machine
 creation, 266–271
 definition, 264, 265
 execution, 272, 273
 vs. SWF, 338–340

sync, 318, 319
tasks, 265
task token mechanism,
320–322
wizard, creation, 267
workflow update, 333–338

T

Task orchestration, 189
Task state
 catch field, 294–296
 parameters, 291, 292
 resource, 290
ResultPath, 297, 298
retrier objects, 292–294
TimeoutSeconds, 296, 297

Timers
 cancel, 251–253
 start, 248–250
Time to Live (TTL), 94, 151

U

Universally unique identifier
(UUID), 132, 164, 276

V

VueJS, 49, 52

W, X, Y, Z

Workflow execution, 208–210