We also want to make HTTP and HTTPS requests from the subnet to the internet. Keep in mind that we have no route to the internet yet. There is no way to access the internet, even with the NACLs. You will change this soon. The following NACLs are needed:

- Allow inbound ephemeral ports from 0.0.0.0/0.
- Allow outbound port 80 (HTTP) to 0.0.0.0/0.
- Allow outbound port 443 (HTTPS) to 0.0.0.0/0.

Find the CloudFormation implementation of the previous NACLs here:

```
NetworkAclEntryInPrivateBackendHTTP:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '10.0.1.0/24'
NetworkAclEntryInPrivateBackendEphemeralPorts:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPrivateBackendHTTP:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPrivateBackendHTTPS:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '110'
    Protocol: '6'
```

Allows inbound HTTP from proxy subnet

Ephemeral ports

Allows outbound HTTP to everywhere

Allows outbound HTTPS to everywhere

```
    PortRange:
      From: '443'
      To: '443'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPrivateBackendEphemeralPorts:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPrivateBackend
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '10.0.1.0/24'
```

Ephemeral ports

5.5.4 Launching virtual machines in the subnets

Your subnets are ready, and you can continue with the EC2 instances. First you describe the proxy, like this:

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Allowing all incoming and outgoing traffic.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - IpProtocol: '-1'
      FromPort: '-1'
      ToPort: '-1'
      CidrIp: '0.0.0.0/0'
    SecurityGroupEgress:
    - IpProtocol: '-1'
      FromPort: '-1'
      ToPort: '-1'
      CidrIp: '0.0.0.0/0'
Proxy:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: 'ami-061ac2e015473fbe2'
    InstanceType: 't2.micro'
    IamInstanceProfile: 'ec2-ssm-core'
    SecurityGroupIds:
    - !Ref SecurityGroup
    SubnetId: !Ref SubnetPublicProxy
    Tags:
    - Key: Name
      Value: Proxy
    UserData: # [...]
  DependsOn: VPCGatewayAttachment
```

This security group allows everything.

Launches in the proxy subnet

We need to manually tell CloudFormation about a dependency here. Without the attachment being created, the instance could launch but without access to the internet.

The private backend has a different configuration, as shown next:

```
Backend:
  Type: 'AWS::EC2::Instance'
  Properties:
    ImageId: 'ami-061ac2e015473fbe2'
    InstanceType: 't2.micro'
    IamInstanceProfile: 'ec2-ssm-core'
    SecurityGroupIds:
    - !Ref SecurityGroup
    SubnetId: !Ref SubnetPrivateBackend
    Tags:
    - Key: Name
      Value: Backend
    UserData:
      'Fn::Base64': |
        #!/bin/bash -ex
        yum -y install httpd
        systemctl start httpd
        echo '<html>...</html>' > /var/www/html/index.html
```

Launches in the private backend subnet

Installs Apache from the internet

Starts the Apache web server

You're now in serious trouble: installing Apache won't work because your private subnet has no route to the internet. Therefore, the `yum install` command will fail, because the public Yum repository is not reachable without access to the internet.

5.5.5 Accessing the internet from private subnets via a NAT gateway

Public subnets have a route to the internet gateway. You can use a similar mechanism to provide outbound internet connectivity for EC2 instances running in private subnets without having a direct route to the internet: create a NAT gateway in a public subnet, and create a route from your private subnet to the NAT gateway. This way, you can reach the internet from private subnets, but the internet can't reach your private subnets. A NAT gateway is a managed service provided by AWS that handles network address translation. Internet traffic from your private subnet will access the internet from the public IP address of the NAT gateway.

Reducing costs for NAT gateway

You have to pay for the traffic processed by a NAT gateway (see VPC Pricing at https://aws.amazon.com/vpc/pricing/ for more details). If your EC2 instances in private subnets will have to transfer huge amounts of data to the internet, you have two options to decrease costs:

- Moving your EC2 instances from the private subnet to a public subnet allows them to transfer data to the internet without using the NAT gateway. Use firewalls to strictly restrict incoming traffic from the internet.
- If data is transferred over the internet to reach AWS services (such as Amazon S3 and Amazon DynamoDB), use gateway VPC endpoints. These endpoints allow your EC2 instances to communicate with S3 and DynamoDB directly and at no additional charge. Furthermore, most other services are accessible from private subnets via interface VPC endpoints (offered by AWS PrivateLink) with an hourly and bandwidth fee.

WARNING NAT gateways are finite resources. A NAT gateway processes up to 100 Gbit/s of traffic and up to 10 million packets per second. A NAT gateway is also bound to an availability zone (introduced in chapter 16).

To keep concerns separated, you'll create a subnet for the NAT gateway as follows:

```
SubnetPublicNAT:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: !Select [0, !GetAZs '']
    CidrBlock: '10.0.0.0/24'
    MapPublicIpOnLaunch: true
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'Public NAT'
RouteTablePublicNAT:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPublicNAT:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicNAT
    RouteTableId: !Ref RouteTablePublicNAT
RoutePublicNATToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePublicNAT
    DestinationCidrBlock: '0.0.0.0/0'
    GatewayId: !Ref InternetGateway
  DependsOn: VPCGatewayAttachment
NetworkAclPublicNAT:
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPublicNAT:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicNAT
    NetworkAclId: !Ref NetworkAclPublicNAT
```

10.0.0.0/24 is the NAT subnet.

The NAT subnet is public with a route to the internet.

We need a bunch of NACL rules to make the NAT gateway work.

To allow all VPC subnets to use the NAT gateway for HTTP and HTTPS, perform the following steps:

1. Allow inbound ports 80 (HTTP) and 443 (HTTPS) from 10.0.0.0/16.
2. Allow outbound ephemeral ports to 10.0.0.0/16.

To allow the NAT gateway to reach out to the internet on HTTP and HTTPS, perform these steps:

- Allow outbound ports 80 (HTTP) and 443 (HTTPS) to 0.0.0.0/0.
- Allow inbound ephemeral ports from 0.0.0.0/0.

Find the CloudFormation implementation of the previous NACL rules next:

```
NetworkAclEntryInPublicNATHTTP:                    <-- Allows inbound HTTP from all VPC subnets
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '10.0.0.0/16'
NetworkAclEntryInPublicNATHTTPS:                   <-- Allows inbound HTTPS from all VPC subnets
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '110'
    Protocol: '6'
    PortRange:
      From: '443'
      To: '443'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '10.0.0.0/16'
NetworkAclEntryInPublicNATEphemeralPorts:          <-- Ephemeral ports
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicNATHTTP:                   <-- Allows outbound HTTP to everywhere
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicNATHTTPS:                  <-- Allows outbound HTTPS to everywhere
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '110'
```

```
    Protocol: '6'
    PortRange:
      From: '443'
      To: '443'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicNATEphemeralPorts:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicNAT
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '10.0.0.0/16'
```

Ephemeral ports

Finally, you can add the NAT gateway itself. The NAT gateway comes with a fixed public IP address (also called an Elastic IP address). All traffic that is routed through the NAT gateway will come from this IP address. We also add a route to the backend's route table to route traffic to 0.0.0.0/0 via the NAT gateway, as shown here:

```
EIPNatGateway:
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: 'vpc'
NatGateway:
  Type: 'AWS::EC2::NatGateway'
  Properties:
    AllocationId: !GetAtt 'EIPNatGateway.AllocationId'
    SubnetId: !Ref SubnetPublicNAT
RoutePrivateBackendToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePrivateBackend
    DestinationCidrBlock: '0.0.0.0/0'
    NatGatewayId: !Ref NatGateway
```

A static public IP address is used for the NAT gateway.

The NAT gateway is placed into the private subnet and associated with the static public IP address.

Route from the Apache subnet to the NAT gateway

Last, one small tweak to the already covered backend EC2 instance is needed to keep dependencies up-to-date, shown here:

```
Backend:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
  DependsOn: RoutePrivateBackendToInternet
```

Helps CloudFormation to understand that the route is needed before the EC2 instance can be created

WARNING The NAT gateway included in the example is not covered by the Free Tier. The NAT gateway will cost you $0.045 per hour and $0.045 per GB of

data processed when creating the stack in the US East (N. Virginia) region. Go to https://aws.amazon.com/vpc/pricing/ to have a look at the current prices.

Now you're ready to create the CloudFormation stack with the template located at https://s3.amazonaws.com/awsinaction-code3/chapter05/vpc.yaml by clicking the CloudFormation Quick-Create Link (http://mng.bz/jmR9). Once you've done so, copy the `ProxyPublicIpAddress` output and open it in your browser. You'll see an Apache test page.

Cleaning up

Don't forget to delete your stack after finishing this section to clean up all used resources. Otherwise, you'll likely be charged for the resources you use.

Summary

- AWS is a shared-responsibility environment in which security can be achieved only if you and AWS work together. You're responsible for securely configuring your AWS resources and your software running on EC2 instances, whereas AWS protects buildings and host systems.
- Keeping your software up-to-date is key and can be automated.
- The Identity and Access Management (IAM) service provides everything needed for authentication and authorization with the AWS API. Every request you make to the AWS API goes through IAM to check whether the request is allowed. IAM controls who can do what in your AWS account. To protect your AWS account, grant only those permissions that your users and roles need.
- Traffic to or from AWS resources like EC2 instances can be filtered based on protocol, port, and source or destination.
- A VPC is a private network in AWS where you have full control. With VPCs, you can control routing, subnets, NACLs, and gateways to the internet or your company network via a VPN. A NAT gateway enables access to the internet from private subnets.
- You should separate concerns in your network to reduce potential damage, for example, if one of your subnets is hacked. Keep every system in a private subnet that doesn't need to be accessed from the public internet, to reduce your attackable surface.

6 Automating operational tasks with Lambda

This chapter covers

- Creating a Lambda function to perform periodic health checks of a website
- Triggering a Lambda function with EventBridge events to automate DevOps tasks
- Searching through your Lambda function's logs with CloudWatch
- Monitoring Lambda functions with CloudWatch alarms
- Configuring IAM roles so Lambda functions can access other services

This chapter is about adding a new tool to your toolbox. The tool we're talking about, AWS Lambda, is as flexible as a Swiss Army knife. You don't need a virtual machine to run your own code anymore, because AWS Lambda offers execution environments for C#/.NET Core, Go, Java, JavaScript/Node.js, Python, and Ruby. All you have to do is implement a function, upload your code, and configure the execution environment. Afterward, your code is executed within a fully managed computing environment. AWS Lambda is well integrated with all parts of AWS,

enabling you to easily automate operations tasks within your infrastructure. We use AWS to automate our infrastructure regularly, such as to add and remove instances to a container cluster based on a custom algorithm and to process and analyze log files.

AWS Lambda offers a maintenance-free and highly available computing environment. You no longer need to install security updates, replace failed virtual machines, or manage remote access (such as SSH or RDP) for administrators. On top of that, AWS Lambda is billed by invocation. Therefore, you don't have to pay for idling resources that are waiting for work (e.g., for a task triggered once a day).

In our first example, you will create a Lambda function that performs periodic health checks for your website. This will teach you how to use the Management Console to get started with AWS Lambda quickly. In our second example, you will learn how to write your own Python code and deploy a Lambda function in an automated way using CloudFormation, which we introduced in chapter 4. Your Lambda function will automatically add a tag to newly launched EC2 instances. At the end of the chapter, we'll show you additional use cases like building web applications and Internet of Things (IoT) backends and processing data with AWS Lambda.

Examples are almost all covered by the Free Tier

The examples in this chapter are mostly covered by the Free Tier. There is one exception: AWS CloudTrail. In case you already created a trail in your account, additional charges—most likely just a few cents—will apply. For details, please see https://aws.amazon.com/cloudtrail/pricing/.

You will find instructions on how to clean up the examples at the end of each section.

You may be asking a more basic question: what exactly is AWS Lambda? Before diving into our first real-world example, let us introduce you, briefly, to Lambda and explain why it's often mentioned in the context of an architecture that's being called *serverless*.

6.1 Executing your code with AWS Lambda

Computing capacity is available at different layers of abstraction on AWS: virtual machines, containers, and functions. You learned about the virtual machines offered by Amazon's EC2 service in chapter 3. Containers offer another layer of abstraction on top of virtual machines; you will learn about containers in chapter 18. AWS Lambda provides computing power, as well, but in a fine-grained manner: it is an execution environment for small functions, rather than a full-blown operating system or container.

6.1.1 What is serverless?

When reading about AWS Lambda, you might have stumbled upon the term *serverless*. In his book *Serverless Architectures on AWS* (Manning, 2022; http://mng.bz/wyr5), Peter Sbarski summarizes the confusion created by this catchy and provocative phrase:

> *[…] the word serverless is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code, or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things.*
>
> —Peter Sbarski

We define a serverless system as one that meets the following criteria:

- No need to manage and maintain virtual machines
- Fully managed service offering scalability and high availability
- Billed per request and by resource consumption

AWS Lambda certainly fits these definitions and is indeed a serverless platform in that AWS handles server configurations and management so the server is essentially invisible to you and takes care of itself. AWS is not the only provider offering a serverless platform. Google (Cloud Functions) and Microsoft (Azure Functions) are other competitors in this area. If you would like to read more about serverless, here is a free chapter from *Serverless Architectures on AWS*, second edition: http://mng.bz/qoEx.

6.1.2 Running your code on AWS Lambda

AWS Lambda is the basic building block of the serverless platform provided by AWS. The first step in the process is to run your code on Lambda instead of on your own server.

As shown in figure 6.1, to execute your code with AWS Lambda, follow these steps:

1. Write the code.
2. Upload your code and its dependencies (such as libraries or modules).
3. Create a function determining the runtime environment and configuration.
4. Invoke the function to execute your code in the cloud.

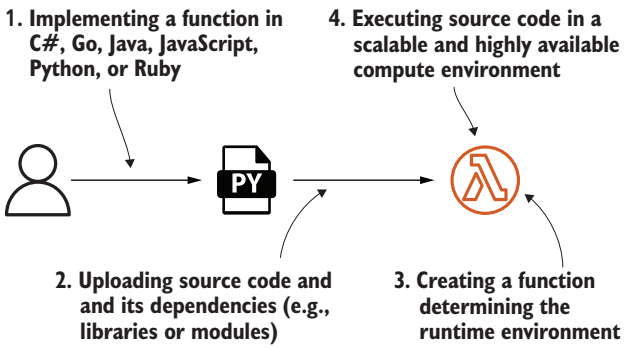


Figure 6.1 Executing code with AWS Lambda

Currently, AWS Lambda offers runtime environments for the following languages:

- C#/.NET Core
- Go
- Java
- JavaScript/Node.js
- Python
- Ruby

Besides that, you can bring your own runtime by using a custom runtime. In theory, a custom runtime supports any programming language. You need to bring your own container image and follow conventions for initializing the function and processing tasks. Doing so adds extra complexity, so we recommend going with one of the available runtimes.

Next, we will compare AWS Lambda with EC2 virtual machines.

6.1.3 Comparing AWS Lambda with virtual machines (Amazon EC2)

What is the difference between using AWS Lambda and virtual machines? First is the granularity of virtualization. Virtual machines provide a full operating system for running one or multiple applications. In contrast, AWS Lambda offers an execution environment for a single function, a small part of an application.

Furthermore, Amazon EC2 offers virtual machines as a service, but you are responsible for operating them in a secure, scalable, and highly available way. Doing so requires you to put a substantial amount of effort into maintenance. In contrast, when building with Lambda, AWS manages the underlying infrastructure for you and provides a production-ready infrastructure.

Beyond that, AWS Lambda is billed per execution, and not per second like when a virtual machine is running. You don't have to pay for unused resources that are waiting for requests or tasks. For example, running a script to check the health of a website every five minutes on a virtual machine would cost you about $3.71 per month. Executing the same health check with AWS Lambda will cost about $0.04 per month.

Table 6.1 compares AWS Lambda and virtual machines in detail. You'll find a discussion of AWS Lambda's limitations at the end of the chapter.

Table 6.1 AWS Lambda compared to Amazon EC2

	AWS Lambda	Amazon EC2
Granularity of virtualization	Small piece of code (a function).	An entire operating system.
Scalability	Scales automatically. A throttling limit prevents you from creating unwanted charges accidentally and can be increased by AWS support if needed.	As you will learn in chapter 17, using an Auto Scaling group allows you to scale the number of EC2 instances serving requests automatically, but configuring and monitoring the scaling activities is your responsibility.

Table 6.1 AWS Lambda compared to Amazon EC2 *(continued)*

	AWS Lambda	Amazon EC2
High availability	Fault tolerant by default. The computing infrastructure spans multiple machines and data centers.	Virtual machines are not highly available by default. Nevertheless, as you will learn in chapter 13, it is possible to set up a highly available infrastructure based on EC2 instances as well.
Maintenance effort	Almost zero. You need only to configure your function.	You are responsible for maintaining all layers between your virtual machine's operating system and your application's runtime environment.
Deployment effort	Almost zero due to a well-defined API	Rolling out your application to a fleet of virtual machines is a challenge that requires tools and know-how.
Pricing model	Pay per request as well as execution time and allocated memory	Pay for operating hours of the virtual machines, billed per second

Looking for limitations and pitfalls of AWS Lambda? Stay tuned: you will find a discussion of Lambda's limitations at the end of the chapter.

That's all you need to know about AWS Lambda to be able to go through the first real-world example. Are you ready?

6.2 Building a website health check with AWS Lambda

Are you responsible for the uptime of a website or application? We do our best to make sure our blog https://cloudonaut.io is accessible 24/7. An external health check acts as a safety net, making sure we, and not our readers, are the first to know when our blog goes down. AWS Lambda is the perfect choice for building a website health check, because you do not need computing resources constantly but only every few minutes for a few milliseconds. This section guides you through setting up a health check for your website based on AWS Lambda.

In addition to AWS Lambda, we are using the Amazon CloudWatch service for this example. Lambda functions publish metrics to CloudWatch by default. Typically, you inspect metrics using charts and create alarms by defining thresholds. For example, a metric could count failures during the function's execution. On top of that, EventBridge provides events that can be used to trigger Lambda functions as well. Here we are using a rule to publish an event every five minutes.

As shown in figure 6.2, your website health check will consist of the following three parts:

1. *Lambda function*—Executes a Python script that sends an HTTP request to your website (e.g., `GET https://cloudonaut.io`) and verifies that the response includes specific text (such as `cloudonaut`).
2. *EventBridge rule*—Triggers the Lambda function every five minutes. This is comparable to the cron service on Linux.

3 *Alarm*—Monitors the number of failed health checks and notifies you via email whenever your website is unavailable.

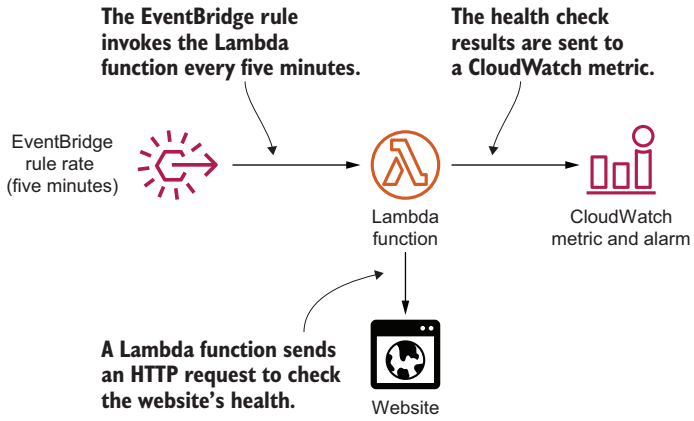


Figure 6.2 The Lambda function performing the website health check is executed every five minutes by a scheduled event. Errors are reported to CloudWatch.

You will use the Management Console to create and configure all the necessary parts manually. In our opinion, this is a simple way to get familiar with AWS Lambda. You will learn how to deploy a Lambda function in an automated way in section 6.3.

6.2.1 Creating a Lambda function

The following step-by-step instructions guide you through setting up a website health check based on AWS Lambda. Open AWS Lambda in the Management Console: https://console.aws.amazon.com/lambda/home. Click Create a Function to start the Lambda function wizard, as shown in figure 6.3.

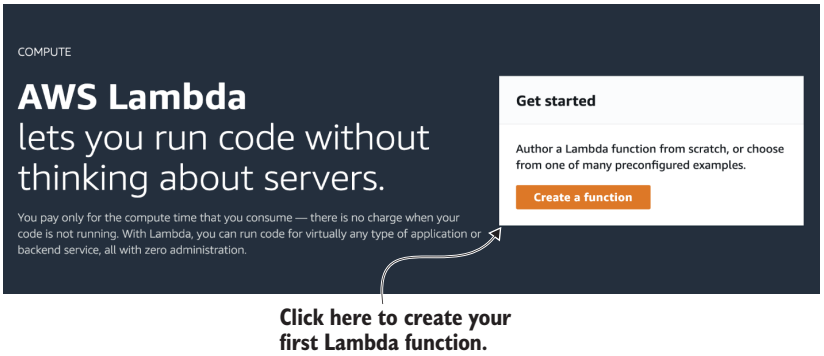


Figure 6.3 Welcome screen: Ready to create your first Lambda function

AWS provides blueprints for various use cases, including the code and the Lambda function configuration. We will use one of these blueprints to create a website health check. Select Use a Blueprint, and search for canary. Next, choose the lambda-canary blueprint. Figure 6.4 illustrates the details.

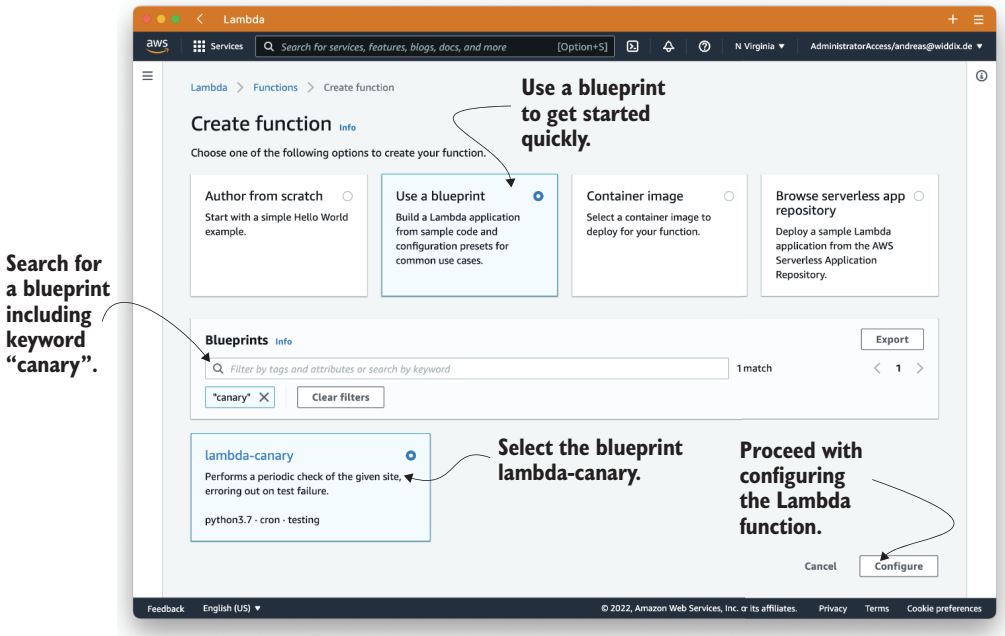


Figure 6.4 Creating a Lambda function based on a blueprint provided by AWS

In the next step of the wizard, you need to specify a name for your Lambda function, as shown in figure 6.5. The function name needs to be unique within your AWS account and the current region US East (N. Virginia). In addition, the name is limited to 64 characters. To invoke a function via the API, you need to provide the function name. Type in website-health-check as the name for your Lambda function.

Continue with creating an IAM role. Select Create a New Role with Basic Lambda Permissions, as shown in figure 6.5, to create an IAM role for your Lambda function. You will learn how your Lambda function uses the IAM role in section 6.3.

Next, configure the scheduled event that will trigger your health check repeatedly. We will use an interval of five minutes in this example. Figure 6.6 shows the settings you need:

1. Select Create a New Rule to create a scheduled event rule.
2. Type in website-health-check as the name for the rule.
3. Enter a description that will help you to understand what is going on if you come back later.

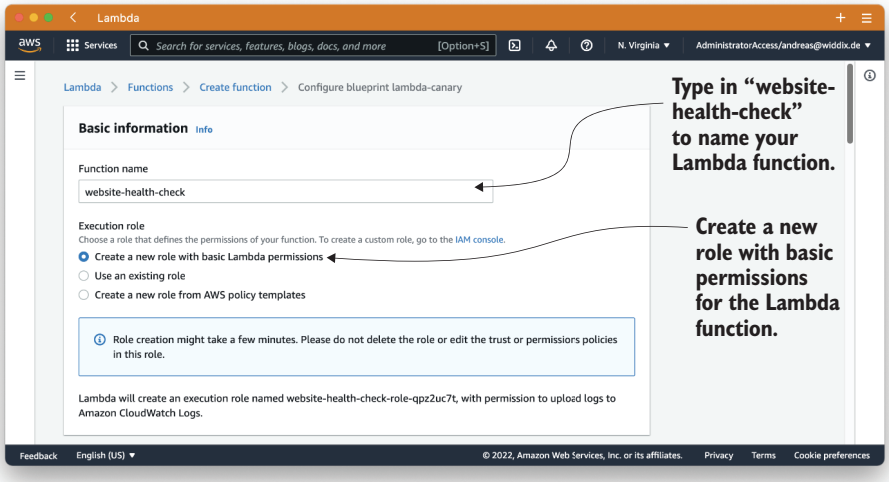


Figure 6.5 Creating a Lambda function: Choose a name and define an IAM role.

4 Select Schedule Expression as the rule type. You will learn about the other option, Event Pattern, at the end of this chapter.
5 Use rate(5 minutes) as the schedule expression.

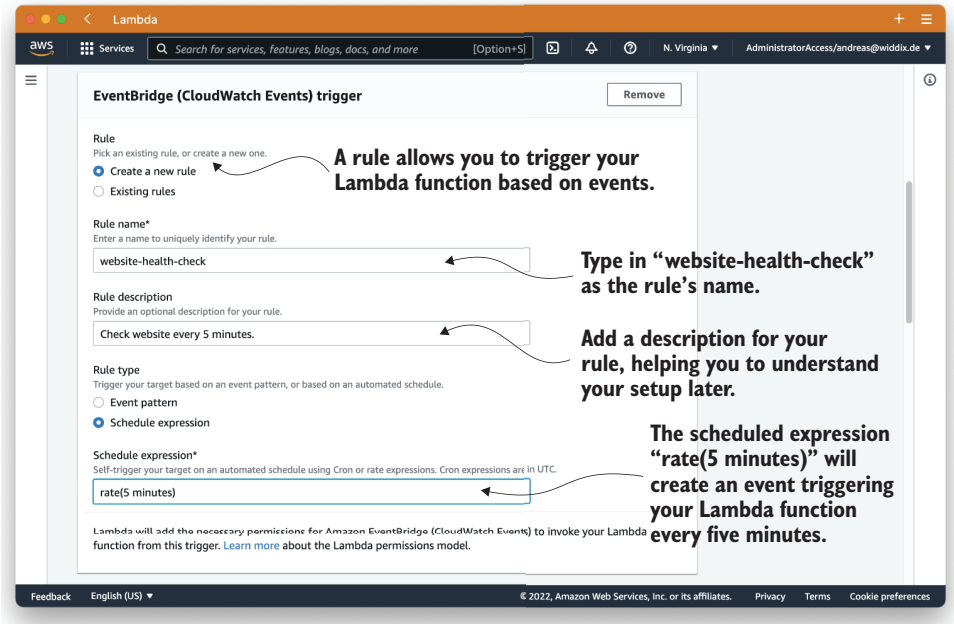


Figure 6.6 Configuring a scheduled event triggering your Lambda function every five minutes

Define recurring tasks using a *schedule expression* in form of `rate($value $unit)`. For example, you could trigger a task every five minutes, every hour, or once a day. `$value` needs to be a positive integer value. Use `minute`, `minutes`, `hour`, `hours`, `day`, or `days` as the unit. For example, instead of triggering the website health check every five minutes, you could use `rate(1 hour)` as the schedule expression to execute the health check every hour. Note that frequencies of less than one minute are not supported.

It is also possible to use the crontab format, shown here, when defining a schedule expression:

```
cron($minutes $hours $dayOfMonth $month $dayOfWeek $year)

 # Invoke a Lambda function at 08:00am (UTC) everyday
 cron(0 8 * * ? *)

 # Invoke a Lambda function at 04:00pm (UTC) every Monday to Friday
 cron(0 16 ? * MON-FRI *)
```

See "Schedule Expressions Using Rate or cron" at http://mng.bz/7ZnQ for more details.

Your Lambda function is missing an integral part: the code. Because you are using a blueprint, AWS has inserted the Python code implementing the website health check for you, as shown in figure 6.7.

The Python code references two environment variables: `site` and `expected`. Environment variables are commonly used to dynamically pass settings to your function. An environment variable consists of a key and a value. Specify the following environment variables for your Lambda function:

- `site`—Contains the URL of the website you want to monitor. Use https://cloudonaut.io if you do not have a website to monitor yourself.
- `expected`—Contains a text snippet that must be available on your website. If the function doesn't find this text, the health check fails. Use `cloudonaut` if you are using https://cloudonaut.io as your website.

The Lambda function is reading the environment variables during its execution, as shown next:

```
SITE = os.environment['site']
 EXPECTED = os.environment['expected']
```

After defining the environment variables for your Lambda function, click the Create Function button at the bottom of the screen.

Congratulations—you have successfully created a Lambda function. Every five minutes, the function is invoked automatically and executes a health check for your website.

You could use this approach to automate recurring tasks, like checking the status of a system, cleaning up unused resources like EBS snapshots, or to send recurring reports.

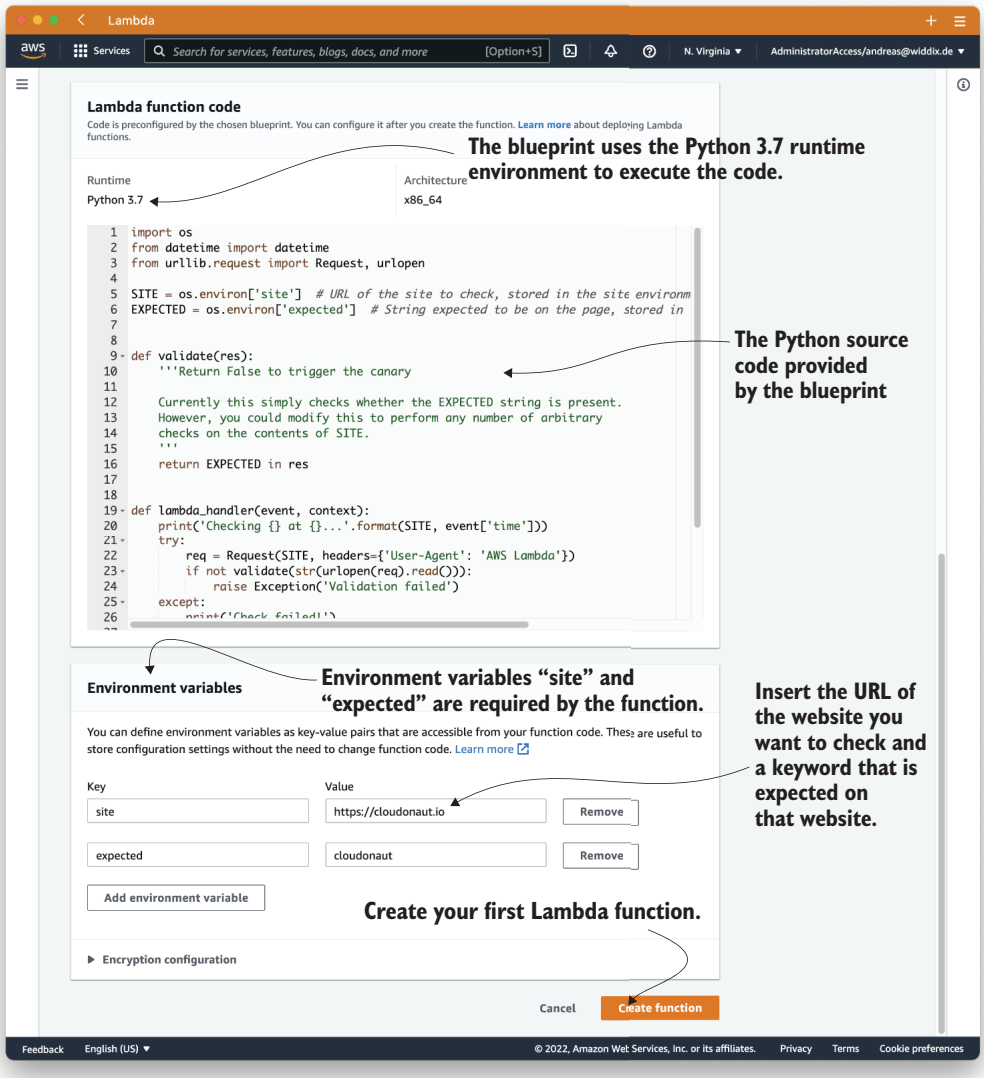


Figure 6.7 The predefined code implementing the website health check and environment variables to pass settings to the Lambda function

So far, you have used a predefined template. Lambda gets much more powerful when you write your own code. But before we look into that, you will learn how to monitor your Lambda function and get notified via email whenever the health check fails.

6.2.2 Use CloudWatch to search through your Lambda function's logs

How do you know whether your website health check is working correctly? How do you even know whether your Lambda function has been executed? It is time to look at

how to monitor a Lambda function. You will learn how to access your Lambda function's log messages first. Afterward, you will create an alarm notifying you if your function fails.

Open the Monitor tab in the details view of your Lambda function. You will find a chart illustrating the number of times your function has been invoked. Reload the chart after a few minutes, in case the chart isn't showing any invocations. To go to your Lambda function's logs, click View Logs in CloudWatch, as shown in figure 6.8.

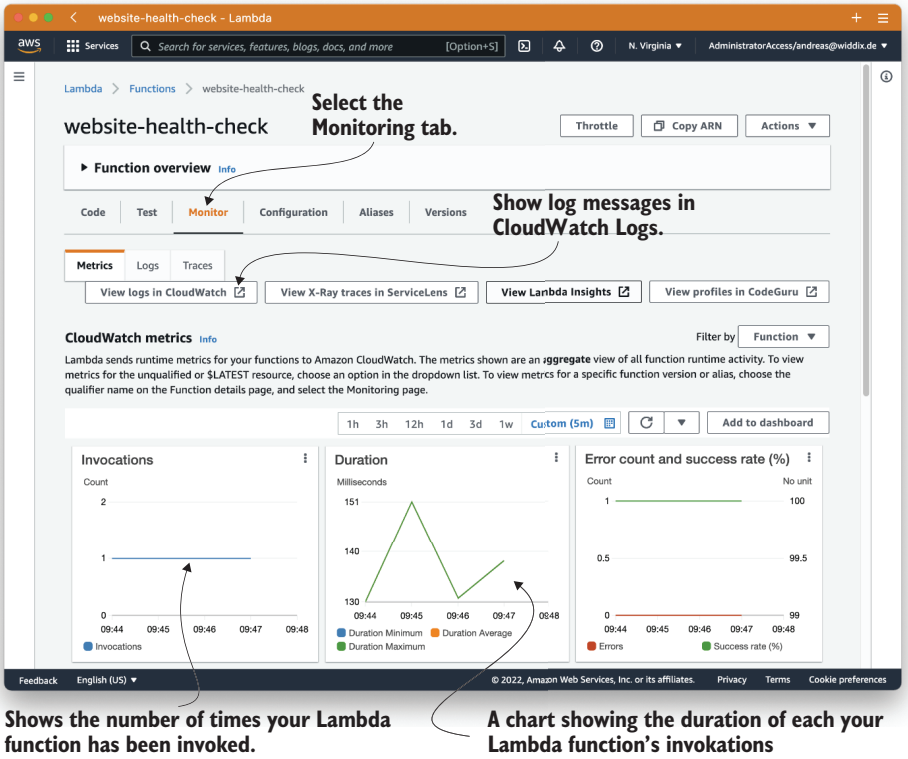


Figure 6.8 Monitoring overview: Get insights into your Lambda function's invocations.

By default, your Lambda function sends log messages to CloudWatch. Figure 6.9 shows the log group named /aws/lambda/website-health-check that was created automatically and collects the logs from your function. Typically, a log group contains multiple log streams, allowing the log group to scale. Click Search Log Group to view the log messages from all streams in one view.

All log messages are presented in the overview of log streams, as shown in figure 6.10. You should be able to find a log message `Check passed!`, indicating that the website health check was executed and passed successfully, for example.

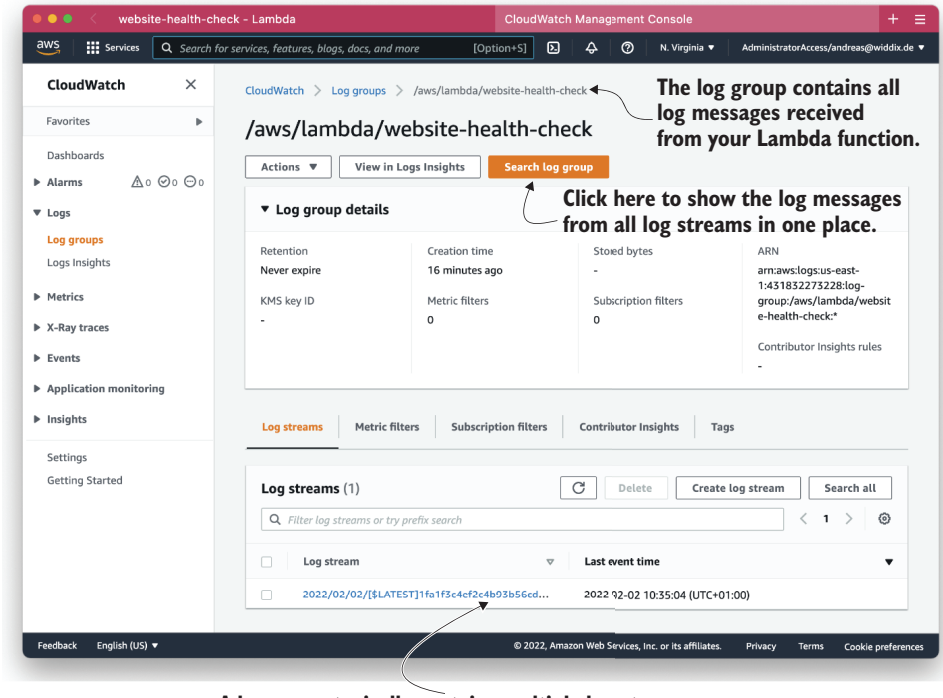


Figure 6.9 A log group collects log messages from a Lambda function stored in multiple log streams.

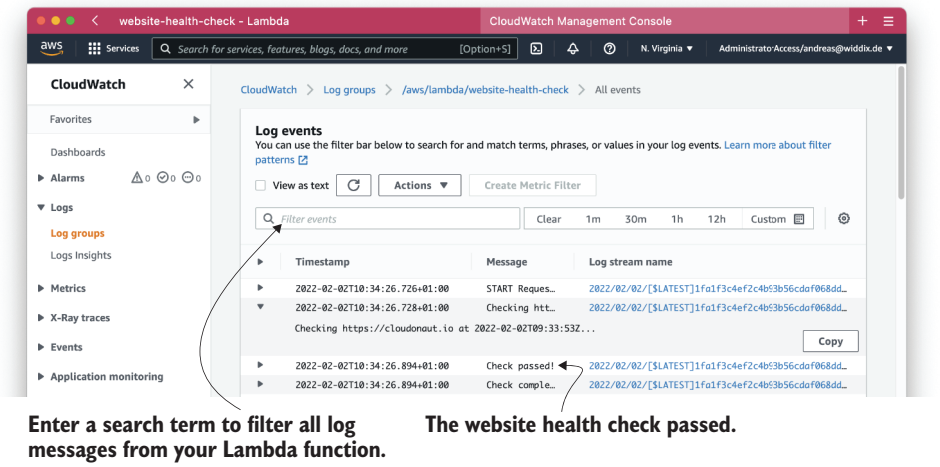


Figure 6.10 CloudWatch shows the log messages of your Lambda function.

The log messages show up after a delay of a few minutes. Reload the table if you are missing any log messages.

Being able to search through log messages in a centralized place is handy when debugging Lambda functions, especially if you are writing your own code. When using Python, you can use `print` statements or use the `logging` module to send log messages to CloudWatch.

6.2.3 Monitoring a Lambda function with CloudWatch metrics and alarms

The Lambda function now checks the health of your website every five minutes, and a log message with the result of each health check is written to CloudWatch. But how do you get notified via email if the health check fails? Each Lambda function publishes metrics to CloudWatch by default. Table 6.2 shows important metrics. Check out http://mng.bz/m298 for a complete list of metrics.

Table 6.2 The CloudWatch metrics published by each Lambda function

Name	Description
Invocations	Counts the number of times a function is invoked. Includes successful and failed invocations.
Errors	Counts the number of times a function failed due to errors inside the function, for example, exceptions or timeouts.
Duration	Measures how long the code takes to run, from the time when the code starts executing to when it stops executing.
Throttles	As discussed at the beginning of the chapter, there is a limit for how many copies of your Lambda function can run at one time. This metric counts how many invocations have been throttled due to reaching this limit. Contact AWS support to increase the limit, if needed.

Whenever the website health check fails, the Lambda function returns an error, which increases the count of the Errors metric. You will create an alarm notifying you via email whenever this metric counts more than zero errors. In general, we recommend creating an alarm on the Errors and Throttles metrics to monitor your Lambda functions.

The following steps guide you through creating a CloudWatch alarm to monitor your website health checks. Your Management Console still shows the CloudWatch service. Select Alarms from the subnavigation menu. Next, click Create Alarm, as shown in figure 6.11.

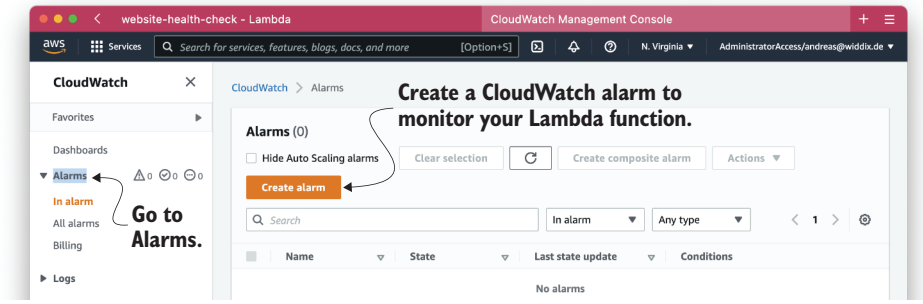


Figure 6.11 Starting the wizard to create a CloudWatch alarm to monitor a Lambda function

The following three steps guide you through the process of selecting the Error metric of your `website-health-check` Lambda function, as illustrated in figure 6.12:

1. Click Select Metric.
2. Choose the Lambda namespace.
3. Select metrics with dimension Function Name.

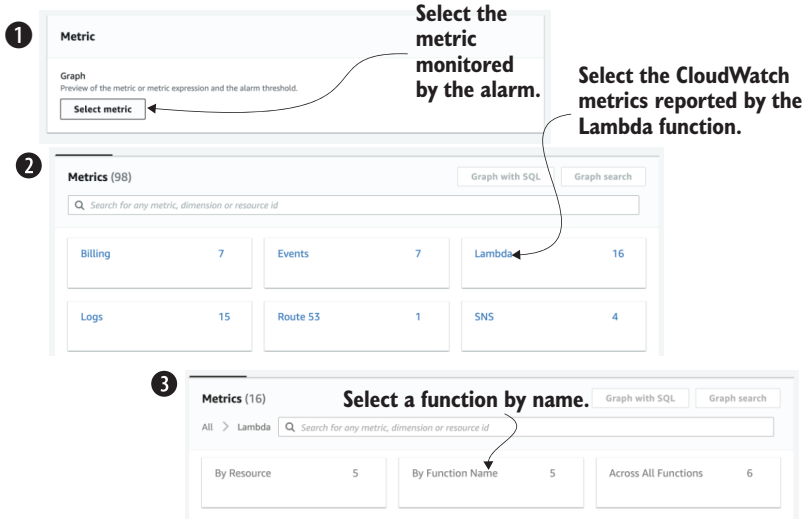


Figure 6.12 Searching the CloudWatch metric to create an alarm

Last but not least, select the Error metric belonging to the Lambbda function `website-health-check` as shown in figure 6.13. Proceed by clicking the Select Metric button.

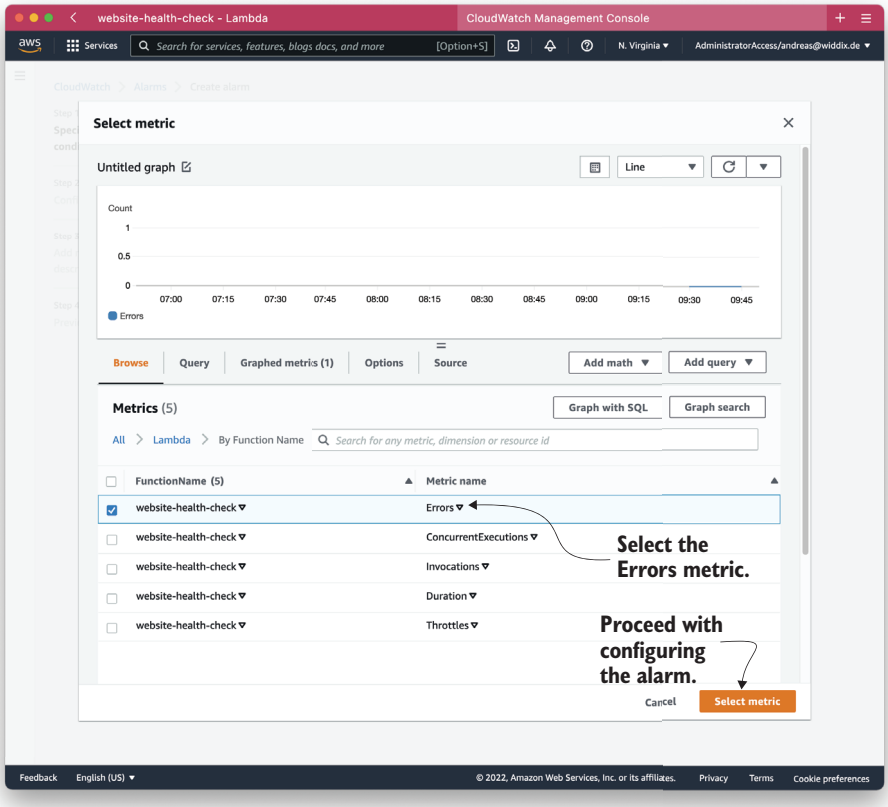


Figure 6.13 Selecting the Error metric of the Lambda function

Next, you need to configure the alarm, as shown in figure 6.14. To do so, specify the statistic, period, and threshold as follows:

1. Choose the statistic Sum to add up all the errors that occurred during the evaluation period.
2. Define an evaluation period of five minutes.
3. Select the Static Threshold option.
4. Choose the Greater operator.
5. Define a threshold of zero.
6. Click the Next button to proceed.

In other words, the alarm state will change to `ALARM` when the Lambda function reports any errors during the evaluation period of five minutes. Otherwise, the alarm state will be `OK`.

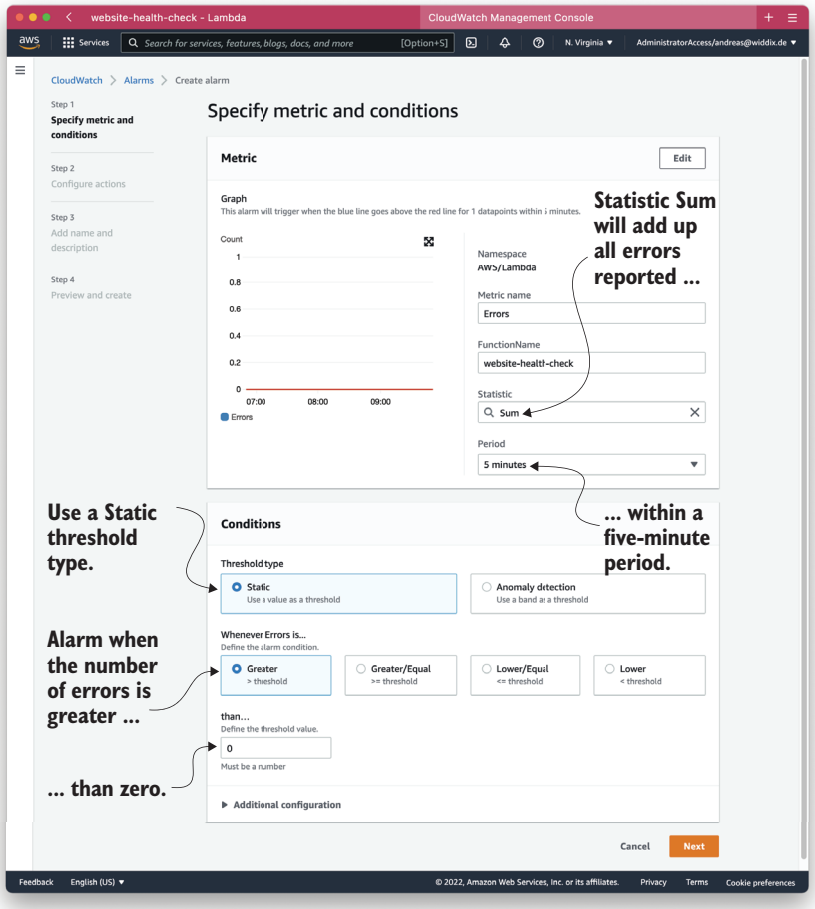


Figure 6.14 Selecting and preparing the metric view for the alarm

The next step, illustrated in figure 6.15, configures the alarm actions so you will be notified via e-mail:

1. Select In Alarm as the state trigger.
2. Create a new SNS topic by choosing Create New Topic.
3. Type in website-health-check as the name for the SNS topic.
4. Enter your email address.
5. Click the Next button to proceed.

In the following step, you need to enter a name for the CloudWatch alarm. Type in website-health-check-error. Afterward, click the Next button to proceed.

After reviewing the configuration, click the Create Alarm button. Shortly after creating the alarm, you will receive an email including a confirmation link for SNS. Check your inbox and click the link to confirm your subscription to the notification list.

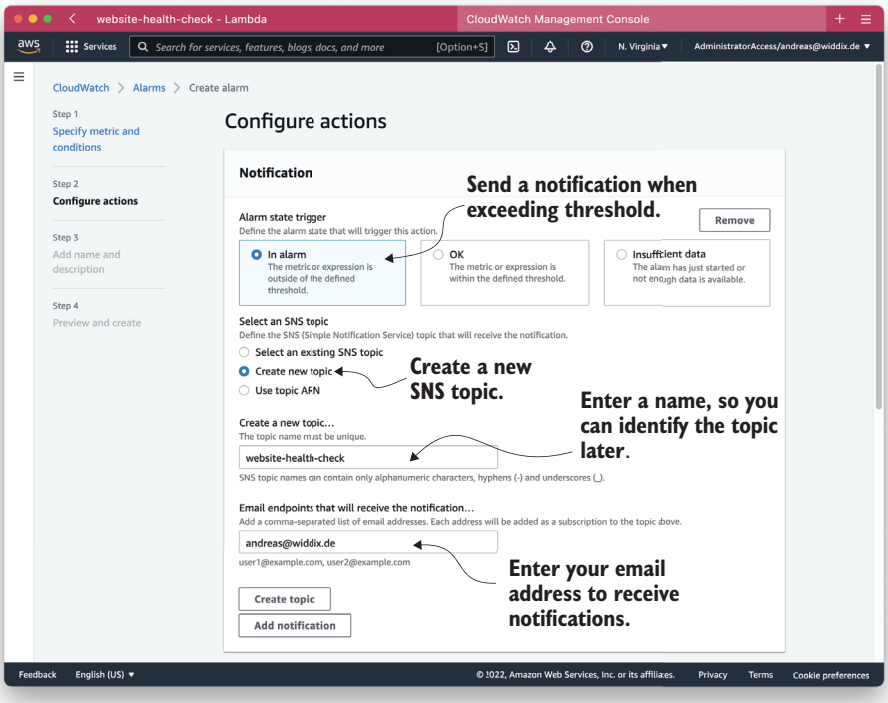


Figure 6.15 Creating an alarm by defining a threshold and defining an alarm action to send notifications via email

To test the alarm, go back to the Lambda function and modify the environment variable `expected`. For example, modify the value from `cloudonaut` to `FAILURE`. This will cause the health check to fail, because the word `FAILURE` does not appear on the website. It might take up to 15 minutes before you receive a notification about the failed website health check via email.

Cleaning up

Open your Management Console and follow these steps to delete all the resources you have created during this section:

1. Go to the AWS Lambda service and delete the function named website-health-check.
2. Open the AWS CloudWatch service, select Logs from the subnavigation options, and delete the log group /aws/lambda/website-health-check.
3. Go to the EventBridge service, select Rules from the subnavigation menu, and delete the rule website-health-check.

4 Open the CloudWatch service, select Alarms from the subnavigation menu, and delete the alarm website-health-check-error.
5 Jump to the AWS IAM service, select Roles from the subnavigation menu, and delete the role whose name starts with website-health-check-role-.
6 Go to the SNS service, select Topics from the subnavigation menu, and delete the rule website-health-check.

6.2.4 Accessing endpoints within a VPC

As illustrated in figure 6.16, by default Lambda functions run outside your networks defined with VPC. However, Lambda functions are connected to the internet and, therefore, are able to access other services. That's exactly what you have been doing when creating a website health check: the Lambda function was sending HTTP requests over the internet.

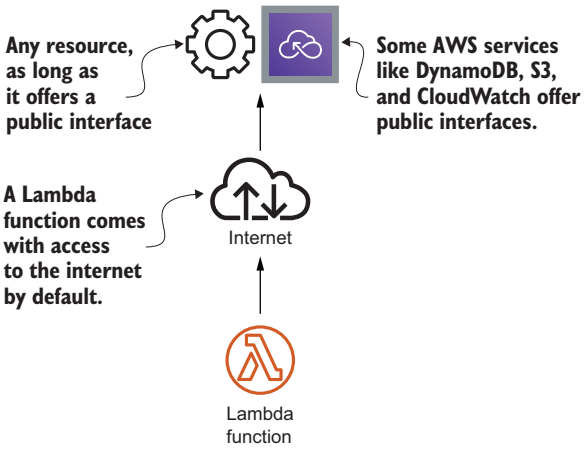


Figure 6.16 By default a Lambda function is connected to the internet and running outside your VPCs.

So, what do you do when you have to reach a resource running in a private network within your VPC, for example, if you want to run a health check for an internal website? If you add network interfaces to your Lambda function, the function can access resources within your VPCs, as shown in figure 6.17.

To do so, you have to define the VPC and the subnets, as well as security groups for your Lambda function. See "Configuring a Lambda Function to Access Resources in an Amazon VPC" at http://mng.bz/5m67 for more details. We have been using the ability to access resources within a VPC to access databases in various projects.

We do recommend connecting a Lambda function to a VPC only when absolutely necessary because it introduces additional complexity. However, being able to connect with resources within your private networks is very interesting, especially when integrating with legacy systems.

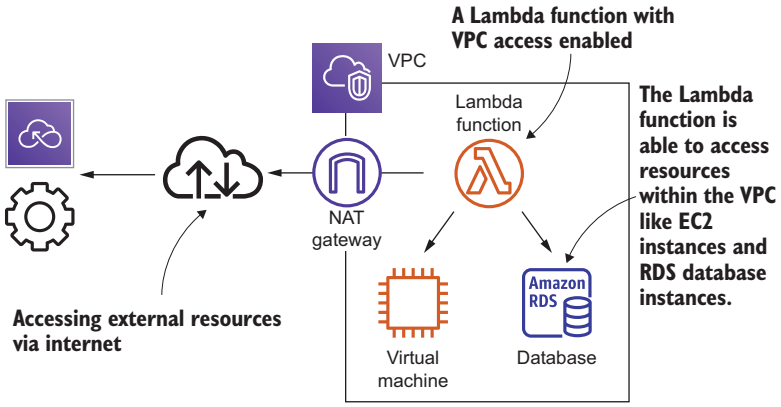


Figure 6.17 Deploying a Lambda function into your VPC allows you to access internal resources (such as database, virtual machines, and so on).

6.3 Adding a tag containing the owner of an EC2 instance automatically

After using one of AWS's predefined blueprints to create a Lambda function, you will implement the function from scratch in this section. We are strongly focused on setting up your cloud infrastructure in an automated way. That's why you will learn how to deploy a Lambda function and all its dependencies without needing the Management Console.

Are you working in an AWS account together with your colleagues? Have you ever wondered who launched a certain EC2 instance? Sometimes you need to find out the owner of an EC2 instance for the following reasons:

- Double-checking whether it is safe to terminate an unused instance without losing relevant data
- Reviewing changes to an instance's configuration with its owner (such as making changes to the firewall configuration)
- Attributing costs to individuals, projects, or departments
- Restricting access to an instance (e.g., so only the owner is allowed to terminate an instance)

Adding a tag that states who owns an instance solves all these use cases. A tag can be added to an EC2 instance or almost any other AWS resource and consists of a key and a value. You can use tags to add information to a resource, filter resources, attribute costs to resources, and restrict access. See "Tag your Amazon EC2 resources" at http://mng.bz/69oR for more details.

It is possible to add tags specifying the owner of an EC2 instance manually. But, sooner or later, someone will forget to add the owner tag. There is a better solution! In the following section, you will implement and deploy a Lambda function that automatically adds a tag containing the name of the user who launched an EC2 instance.

But how do you execute a Lambda function every time an EC2 instance is launched so that you can add the tag?

6.3.1 Event-driven: Subscribing to EventBridge events

EventBridge is an event bus used by AWS, third-party vendors, and customers like you, to publish and subscribe to events. In this example, you will create a rule to listen for events from AWS. Whenever something changes in your infrastructure, an event is generated in near real time and the following things occur:

- CloudTrail emits an event for every call to the AWS API if CloudTrail is enabled in the AWS account and region.
- EC2 emits events whenever the state of an EC2 instances changes (such as when the state changes from `Pending` to `Running`).
- AWS emits an event to notify you of service degradations or downtimes.

Whenever you launch a new EC2 instance, you are sending a call to the AWS API. Subsequently, CloudTrail sends an event to EventBridge. Our goal is to add a tag to every new EC2 instance. Therefore, we are executing a function for every event that indicates the launch of a new EC2 instance. To trigger a Lambda function whenever such an event occurs, you need a rule. As illustrated in figure 6.18, the rule matches incoming events and routes them to a target, a Lambda function in our case.

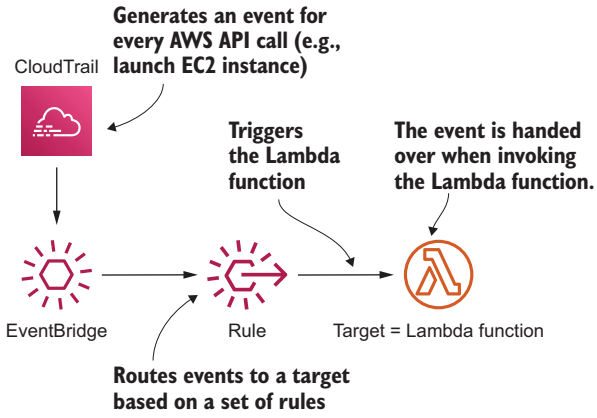


Figure 6.18 CloudTrail generates an event for every AWS API call; a rule routes the event to the Lambda function.

Listing 6.1 shows some of the event details generated by CloudTrail whenever someone launches an EC2 instance. For our case, we're interested in the following information:

- `detail-type`—The event has been created by CloudTrail.
- `source`—The EC2 service is the source of the event.
- `eventName`—The event name `RunInstances` indicates that the event was generated because of an AWS API call launching an EC2 instance.
- `userIdentity`—Who called the AWS API to launch an instance?

- `responseElements`—The response from the AWS API when launching an instance. This includes the ID of the launched EC2 instance; we will need to add a tag to the instance later.

Listing 6.1 Event generated by CloudTrail when launching an EC2 instance

```
{
  "version": "0",
  "id": "2db486ef-6775-de10-1472-ecc242928abe",
  "detail-type": "AWS API Call via CloudTrail",
  "source": "aws.ec2",
  "account": "XXXXXXXXXXXX",
  "time": "2022-02-03T11:42:25Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "eventVersion": "1.08",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "XXXXXXXXXXXX",
      "arn": "arn:aws:iam::XXXXXXXXXXXX:user/myuser",
      "accountId": "XXXXXXXXXXXX",
      "accessKeyId": "XXXXXXXXXXXX",
      "userName": "myuser"
    },
    "eventTime": "2022-02-03T11:42:25Z",
    "eventSource": "ec2.amazonaws.com",
    "eventName": "RunInstances",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "109.90.107.17",
    "userAgent": "aws-cli/2.4.14 Python/3.9.10 Darwin/21.2.0
      ➥ source/arm64 prompt/off command/ec2.run-instances",
    "requestParameters": {
      [...]
    },
    "responseElements": {
      "requestId": "d52b86c7-5bf8-4d19-86e8-112e59164b21",
      "reservationId": "r-08131583e8311879d",
      "ownerId": "166876438428",
      "groupSet": {},
      "instancesSet": {
        "items": [
          {
            "instanceId": "i-07a3c0d78dc1cb505",
            "imageId": "ami-01893222c83843146",
            [...]
          }
        ]
      }
    },
    "requestID": "d52b86c7-5bf8-4d19-86e8-112e59164b21",
    "eventID": "8225151b-3a9c-4275-8b37-4a317dfe9ee2",
    "readOnly": false,
    "eventType": "AwsApiCall",
```

CloudTrail generated the event.

Someone sent a call to the AWS API, affecting the EC2 service.

Information about the user who launched the EC2 instance

ID of the user who launched the EC2 instance

Event was generated because a RunInstances call (used to launch an EC2 instance) was processed by the AWS API.

Response of the AWS API when launching the instance

ID of the launched EC2 instance

```
    "managementEvent": true,
    "recipientAccountId": "XXXXXXXXXXXX",
    "eventCategory": "Management",
    "tlsDetails": {
      [...]
    }
  }
}
```

A rule consists of an event pattern for selecting events, along with a definition of one or multiple targets. The following pattern selects all events from CloudTrail generated by an AWS API call affecting the EC2 service. The pattern matches four attributes from the event described in the next listing: `source`, `detail-type`, `eventSource`, and `eventName`.

Listing 6.2 The pattern to filter events from CloudTrail

```
{
  "source": [
    "aws.ec2"
  ],
  "detail-type": [
    "AWS API Call via CloudTrail"
  ],
  "detail": {
    "eventSource": [
      "ec2.amazonaws.com"
    ],
    "eventName": [
      "RunInstances"
    ]
  }
}
```

Filters events from the EC2 service

Filters events from CloudTrail caused by AWS API calls

Filters events from the EC2 service

Filters events with event name RunInstances, which is the AWS API call to launch an EC2 instance

Defining filters on other event attributes is possible as well, in case you are planning to write another rule in the future. The rule format stays the same.

When specifying an event pattern, we typically use the following fields, which are included in every event:

- `source`—The namespace of the service that generated the event. See "Amazon Resource Names (ARNs)" at http://mng.bz/o5WD for details.
- `detail-type`—Categorizes the event in more detail.

See "EventBridge Event Patterns" at http://mng.bz/nejd for more detailed information.

You have now defined the events that will trigger your Lambda function. Next, you will implement the Lambda function.

6.3.2 Implementing the Lambda function in Python

Implementing the Lambda function to tag an EC2 instance with the owner's user name is simple. You will need to write no more than 10 lines of Python code. The programming

model for a Lambda function depends on the programming language you choose. Although we are using Python in our example, you will be able to apply what you've learned when implementing a Lambda function in C#/.NET Core, Go, Java, JavaScript/Node.js, Python, or Ruby. As shown in the next listing, your function written in Python needs to implement a well-defined structure.

Listing 6.3 Lambda function written in Python

The name of the Python function, which is referenced by the AWS Lambda as the function handler. The event parameter is used to pass the CloudWatch event, and the context parameter includes runtime information.

```
def lambda_handler(event, context):
    # Insert your code
    return
```

It is your job to implement the function.

Use return to end the function execution. It is not useful to hand over a value in this scenario, because the Lambda function is invoked asynchronously by a CloudWatch event.

> **Where is the code located?**
>
> As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code3. Switch to the chapter06 directory, which includes all files needed for this example.

Time to write some Python code! Listing 6.4 for `lambda_function.py` shows the function, which receives an event from CloudTrail indicating that an EC2 instance has been launched recently, and adds a tag including the name of the instance's owner. The AWS SDK for Python 3.9, named `boto3`, is provided out of the box in the Lambda runtime environment for Python 3.9. In this example, you are using the AWS SDK to create a tag for the EC2 instance `ec2.create_tags(...)`. See the Boto3 documentation at https://boto3.readthedocs.io/en/latest/index.html if you are interested in the details of `boto3`.

Listing 6.4 Lambda function adding a tag to EC2 instance

Creates an AWS SDK client for EC2

The name of the function used as entry point for the Lambda function

Logs the incoming event for debugging

Extracts the user's name from the CloudTrail event

```
import boto3
ec2 = boto3.client('ec2')

def lambda_handler(event, context):
  print(event)
  if "/" in event['detail']['userIdentity']['arn']:
      userName = event['detail']['userIdentity']['arn'].split('/')[1]
  else:
      userName = event['detail']['userIdentity']['arn']
  instanceId = event['detail']['responseElements']['instancesSet']
```

```
    ➥ ['items'][0]['instanceId']
    print("Adding owner tag " + userName + " to instance " + instanceId + ".")
    ec2.create_tags(Resources=[instanceId,],
    ➥ Tags=[{'Key': 'Owner', 'Value': userName},])
    return
```

Extracts the instance's ID from the CloudTrail event

Adds a tag to the EC2 instance using the key owner and the user's name as value

After implementing your function in Python, the next step is to deploy the Lambda function with all its dependencies.

6.3.3 Setting up a Lambda function with the Serverless Application Model (SAM)

You have probably noticed that we are huge fans of automating infrastructures with CloudFormation. Using the Management Console is a perfect way to take the first step when learning about a new service on AWS. But leveling up from manually clicking through a web interface to fully automating the deployment of your infrastructure should be your second step.

AWS released the *Serverless Application Model* (SAM) in 2016. SAM provides a framework for serverless applications, extending plain CloudFormation templates to make it easier to deploy Lambda functions. The next listing shows how to define a Lambda function using SAM and a CloudFormation template.

Listing 6.5 Defining a Lambda function with SAM within a CloudFormation template

A special resource provided by SAM allows us to define a Lambda function in a simplified way. CloudFormation will generate multiple resources out of this declaration during the transformation phase.

The CloudFormation template version, not the version of your code

Transforms are used to process your template. We're using the SAM transformation.

```
---
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Adding an owner tag to EC2 instances automatically
Resources:
  # [...]
  EC2OwnerTagFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: lambda_function.lambda_handler
      Runtime: python3.9
      Architectures:
      - arm64
      CodeUri: '.'
      Policies:
      - Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Action: 'ec2:CreateTags'
          Resource: '*'
      Events:
```

The handler is a combination of your script's filename and Python function name.

Uses the Python 3.9 runtime environment

Choose the ARM architecture for better price performance.

The current directory will be bundled, uploaded, and deployed. You will learn more about that soon.

Authorizes the Lambda function to call other AWS services (more on that next)

The definition of the event invoking the Lambda function

```
EventBridgeRule:
  Type: EventBridgeRule
  Properties:
    Pattern:
      source:
      - 'aws.ec2'
      detail-type:
      - 'AWS API Call via CloudTrail'
      detail:
        eventSource:
        - 'ec2.amazonaws.com'
        eventName:
        - 'RunInstances'
```

Creates an EventBridge rule

Configures the filter pattern

Please note: this example uses CloudTrail, which records all the activity within your AWS account. The CloudFormation template creates a trail to store an audit log on S3. That's needed because the EventBridge rule does not work without an active trail.

6.3.4 Authorizing a Lambda function to use other AWS services with an IAM role

Lambda functions typically interact with other AWS services. For instance, they write log messages to CloudWatch allowing you to monitor and debug your Lambda function. Or they create a tag for an EC2 instance, as in the current example. Therefore, calls to the AWS APIs need to be authenticated and authorized. Figure 6.19 shows a Lambda function assuming an IAM role to be able to send authenticated and authorized requests to other AWS services.

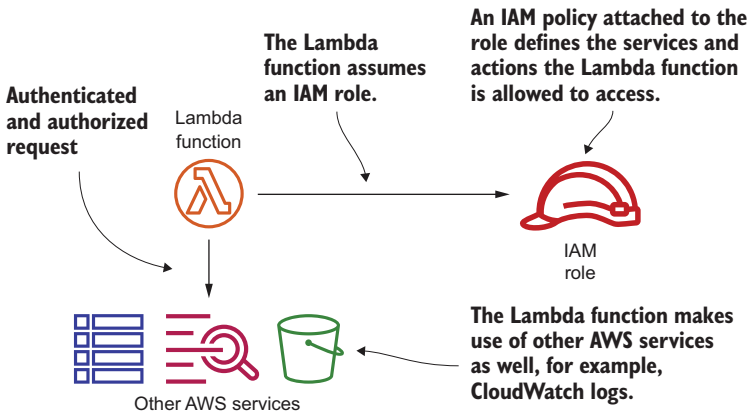


Figure 6.19 A Lambda function assumes an IAM role to authenticate and authorize requests to other AWS services.

Temporary credentials are generated based on the IAM role and injected into each invocation via environment variables (such as `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`). Those environment variables are used by the AWS SDK to sign requests automatically.

You should follow the least-privilege principle: your function should be allowed to access only services and actions that are needed to perform the function's task. You should specify a detailed IAM policy granting access to specific actions and resources.

Listing 6.6 shows an excerpt from the Lambda function's CloudFormation template based on SAM. When using SAM, an IAM role is created for each Lambda function by default. A managed policy that grants write access to CloudWatch logs is attached to the IAM role by default as well. Doing so allows the Lambda function to write to CloudWatch logs.

So far the Lambda function is not allowed to create a tag for the EC2 instance. You need a custom policy granting access to the `ec2:CreateTags`.

Listing 6.6 A custom policy for adding tags to EC2 instances

```
# [...]
 EC2OwnerTagFunction:
   Type: AWS::Serverless::Function
   Properties:
     Handler: lambda_function.lambda_handler
     Runtime: python3.9
     CodeUri: '.'
     Policies:
     - Version: '2012-10-17'
       Statement:
       - Effect: Allow
         Action: 'ec2:CreateTags'
         Resource: '*'
     # [...]
```

Defines a custom IAM policy that will be attached to the Lambda function's IAM role

The statement allows …

…creating tags…

…for all resources.

If you implement another Lambda function in the future, make sure you create an IAM role granting access to all the services your function needs to access (e.g., reading objects from S3, writing data to a DynamoDB database). Revisit section 5.3 if you want to recap the details of IAM.

6.3.5 Deploying a Lambda function with SAM

To deploy a Lambda function, you need to upload the deployment package to S3. The deployment package is a zip file including your code as well as additional modules. Afterward, you need to create and configure the Lambda function as well as all the dependencies (the IAM role, event rule, and so on). Using SAM in combination with the AWS CLI allows you to accomplish both tasks.

First, you need to create an S3 bucket to store your deployment packages. Use the following command, replacing `$yourname` with your name to avoid name conflicts with other readers:

```
$ aws s3 mb s3://ec2-owner-tag-$yourname
```

Execute the following command to create a deployment package and upload the package to S3. Please note: the command creates a copy of your template at output.yaml,

with a reference to the deployment package uploaded to S3. Make sure your working directory is the code directory chapter06 containing the template.yaml and lambda_function.py files:

```
$ aws cloudformation package --template-file template.yaml \
➥ --s3-bucket ec2-owner-tag-$yourname --output-template-file output.yaml
```

By typing the following command in your terminal, you are deploying the Lambda function. This results in a CloudFormation stack named `ec2-owner-tag`:

```
$ aws cloudformation deploy --stack-name ec2-owner-tag \
➥ --template-file output.yaml --capabilities CAPABILITY_IAM
```

You are a genius! Your Lambda function is up and running. Launch an EC2 instance, and you will find a tag with your username `myuser` attached after a few minutes.

Cleaning up

If you have launched an EC2 instance to test your Lambda function, don't forget to terminate the instance afterward. Otherwise, it is quite simple to delete the Lambda function and all its dependencies. Just execute the following command in your terminal. Replace `$yourname` with your name:

```
$ CURRENT_ACCOUNT=$(aws sts get-caller-identity --query Account \
➥ --output text)
$ aws s3 rm --recursive s3://ec2-owner-tag-${CURRENT_ACCOUNT}/
$ aws cloudformation delete-stack --stack-name ec2-owner-tag
$ aws s3 rb s3://ec2-owner-tag-$yourname --force
```

6.4 What else can you do with AWS Lambda?

In the last part of the chapter, we would like to share what else is possible with AWS Lambda, starting with Lambda's limitations and insights into the serverless pricing model. We will end with three use cases for serverless applications we have built for our consulting clients.

6.4.1 What are the limitations of AWS Lambda?

Executing a Lambda function cannot exceed 15 minutes. This means the problem you are solving with your function needs to be small enough to fit into the 900-second limit. It is probably not possible to download 10 GB of data from S3, process the data, and insert parts of the data into a database within a single invocation of a Lambda function. But even if your use case fits into the 900-second constraint, make sure that it will fit under all circumstances. Here's a short anecdote from one of our first serverless projects: we built a serverless application that preprocessed analytics data from news sites. The Lambda functions typically processed the data within less than 180 seconds.

But when the 2017 US elections came, the volume of the analytics data exploded in a way no one expected. Our Lambda functions were no longer able to complete within 300 seconds—which was the maximum back then. It was a show-stopper for our serverless approach.

AWS Lambda provisions and manages the resources needed to run your function. A new execution context is created in the background every time you deploy a new version of your code, go a long time without any invocations, or when the number of concurrent invocations increases. Starting a new execution context requires AWS Lambda to download your code, initialize a runtime environment, and load your code. This process is called a *cold start*. Depending on the size of your deployment package, the runtime environment, and your configuration, a cold start could take from a few milliseconds to a few seconds.

In many scenarios, the increased latency caused by a cold start is not a problem at all. The examples demonstrated in this chapter—a website health check and automated tags for EC2 instances—are not negatively affected by a cold start. To minimize cold-start times, you should keep the size of your deployment package as small as possible, provision additional memory, and use a runtime environment like JavaScript/Node.js or Python.

However, when processing real-time data or user interactions, a cold start is undesirable. For those scenarios, you could enable provisioned concurrency for a Lambda function. With provisioned concurrency, you tell AWS to keep a certain amount of execution contexts warm, even when the Lambda function is not processing any requests. As long as the provisioned concurrency exceeds the required number of execution contexts, you will not experience cold starts. The downside is you pay $0.0000041667 for every provisioned GB per second, whether or not you use the capacity. However, you will get a discount on the cost incurred for the actual term of the Lambda function.

Another limitation is the maximum amount of memory you can provision for a Lambda function: 10,240 MB. If your Lambda function uses more memory, its execution will be terminated.

It is also important to know that CPU and networking capacity are allocated to a Lambda function based on the provisioned memory as well. So, if you are running computing- or network-intensive work within a Lambda function, increasing the provisioned memory will probably improve performance.

At the same time, the default limit for the maximum size of the compressed deployment package (zip file) is 250 MB. When executing your Lambda function, you can use up to 512 MB nonpersistent disk space mounted to `/tmp`. Look at "Lambda Quotas" at http://mng.bz/vXda if you want to learn more about Lambda's limitations.

6.4.2 Effects of the serverless pricing model

When launching a virtual machine, you have to pay AWS for every operating hour, billed in second intervals. You are paying for the machines whether or not you are

using the resource they provide. Even when nobody is accessing your website or using your application, you are paying for the virtual machine.

That's totally different with AWS Lambda. Lambda is billed per request. Costs occur only when someone accesses your website or uses your application. That's a game changer, especially for applications with uneven access patterns or for applications that are used rarely. Table 6.3 explains the Lambda pricing model in detail. Please note: when creating a Lambda function, you select the architecture. As usual, the Arm architecture based on AWS Graviton is the cheaper option.

Table 6.3 AWS Lambda pricing model

	Free Tier	x86	Arm
Number of Lambda function invocations	First 1 million requests every month are free.	$0.0000002 per request	$0.0000002 per request
Duration billed in 1 ms increments based on the amount of memory you provisioned for your Lambda function	Using the equivalent of 400,000 seconds of a Lambda function with 1 GB is free of charge provisioned memory every month.	$0.0000166667 for using 1 GB for one second	$0.0000133334 for using 1 GB for one second

Free Tier for AWS Lambda

The Free Tier for AWS Lambda does not expire after 12 months. That's a huge difference compared to the Free Tier of other AWS services (such as EC2) where you are eligible for the Free Tier only within the first 12 months, after creating an AWS account.

Sounds complicated? Figure 6.20 shows an excerpt of an AWS bill. The bill is from November 2017 and belongs to an AWS account we are using to run a chatbot (see https://marbot.io). Our chatbot implementation is 100% serverless. The Lambda functions were executed 1.2 million times in November 2017, which results in a charge of $0.04. All our Lambda functions are configured to provision 1536 MB memory. In total, all our Lambda functions have been running for 216,000 seconds, or around 60 hours, in November 2017. That's still within the Free Tier of 400,000 seconds with 1 GB provisioned memory every month. So, in total we had to pay $0.04 for using AWS Lambda in November 2017, which allowed us to serve around 400 customers with our chatbot.

This is only a small piece of our AWS bill, because other services we used together with AWS Lambda—for example, to store data—add more significant costs to our bill.

Don't forget to compare costs between AWS Lambda and EC2. Especially in a high-load scenario with more than 10 million requests per day, using AWS Lambda will probably result in higher costs compared to using EC2. But comparing infrastructure costs is only one part of what you should be looking at. Consider the total cost of

▾ Lambda		$0.04
▾ US East (Northern Virginia) Region		$0.04
AWS Lambda Lambda-GB-Second		$0.00
AWS Lambda - Compute Free Tier - 400,000 GB-Seconds - US East (Northern Virginia)	331,906.500 seconds	$0.00
AWS Lambda Request		$0.04
0.0000000367 USD per AWS Lambda - Requests Free Tier - 1,000,000 Requests - US East (Northern Virginia) (blended price)*	1,209,096 Requests	$0.04

Figure 6.20 Excerpt from our AWS bill from November 2017 showing costs for AWS Lambda

ownership (TOC), including costs for managing your virtual machines, performing load and resilience tests, and automating deployments. Our experience has shown that the total cost of ownership is typically lower when running an application on AWS Lambda compared to Amazon EC2.

The last part of the chapter focuses on additional use cases for AWS Lambda besides automating operational tasks, as you have done thus far.

6.4.3 Use case: Web application

A common use case for AWS Lambda is building a backend for a web or mobile application. As illustrated in figure 6.21, an architecture for a serverless web application typically consists of the following building blocks:

- *Amazon API Gateway*—Offers a scalable and secure REST API that accepts HTTPS requests from your web application's frontend or mobile application.
- *AWS Lambda*—Lambda functions are triggered by the API gateway. Your Lambda function receives data from the request and returns the data for the response.

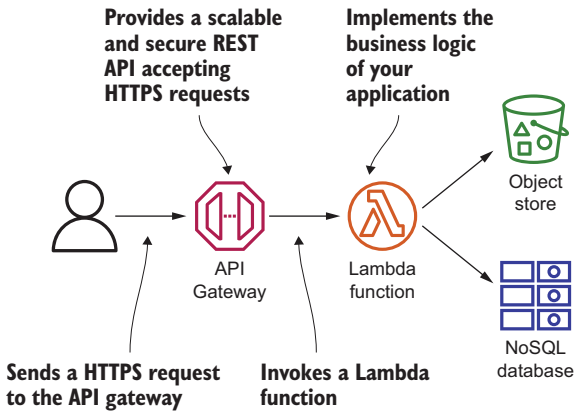


Figure 6.21 A web application build with API Gateway and Lambda

- *Object store and NoSQL database*—For storing and querying data, your Lambda functions typically use additional services offering object storage or NoSQL databases, for example.

Do you want to get started building web applications based on AWS Lambda? We recommend *AWS Lambda in Action* from Danilo Poccia (Manning, 2016).

6.4.4 Use case: Data processing

Another popular use case for AWS Lambda follows: event-driven data processing. Whenever new data is available, an event is generated. The event triggers the data processing needed to extract or transform the data. Figure 6.22 shows an example:

1. The load balancer collects access logs and uploads them to an object store periodically.
2. Whenever an object is created or modified, the object store triggers a Lambda function automatically.
3. The Lambda function downloads the file, including the access logs, from the object store and sends the data to an Elasticsearch database to be available for analytics.

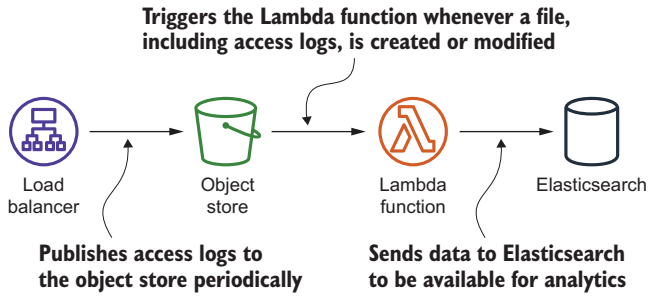


Figure 6.22 Processing access logs from a load balancer with AWS Lambda

We have successfully implemented this scenario in various projects. Keep in mind the maximum execution limit of 900 seconds when implementing data-processing jobs with AWS Lambda.

6.4.5 Use case: IoT backend

The AWS IoT service provides building blocks needed to communicate with various devices (things) and build event-driven applications. Figure 6.23 shows an example. Each thing publishes sensor data to a message broker. A rule filters the relevant messages and triggers a Lambda function. The Lambda function processes the event and decides what steps are needed based on the business logic you provide.

We built a proof of concept for collecting sensor data and publishing metrics to a dashboard with AWS IoT and AWS Lambda, for example.

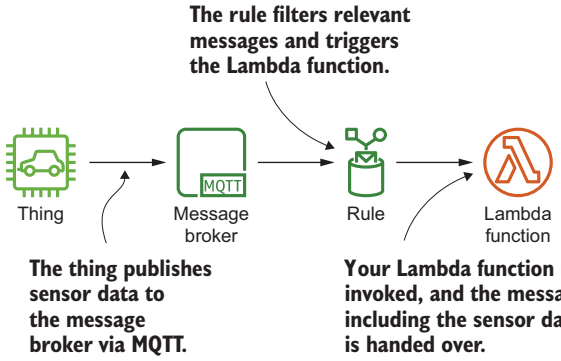


Figure 6.23 Processing events from an IoT device with AWS Lambda

We have gone through three possible use cases for AWS Lambda, but we haven't covered all of them. AWS Lambda is integrated with many other services as well. If you want to learn more about AWS Lambda, we recommend the following books:

- *AWS Lambda in Action* by Danilo Poccia (Manning, 2016) is an example-driven tutorial that teaches how to build applications using an event-based approach on the backend.
- *Serverless Architectures on AWS*, second edition, by Peter Sbarski (Manning, 2022) teaches how to build, secure, and manage serverless architectures that can power the most demanding web and mobile apps.

Summary

- AWS Lambda allows you to run your C#/.NET Core, Go, Java, JavaScript/Node.js, Python and Ruby code within a fully managed, highly available, and scalable environment.
- The Management Console and blueprints offered by AWS help you to get started quickly.
- By using a schedule expression, you can trigger a Lambda function periodically. This is comparable to triggering a script with the help of a cron job.
- CloudWatch is the place to go when it comes to monitoring and debugging your Lambda functions.
- The Serverless Application Model (SAM) enables you to deploy a Lambda function in an automated way with AWS CloudFormation.
- Many event sources exist for using Lambda functions in an event-driven way. For example, you can subscribe to events triggered by CloudTrail for every request you send to the AWS API.
- The most important limitation of a Lambda function is the maximum duration of 900 seconds per invocation.
- AWS Lambda can be used to built complex services as well, from typical web applications, to data analytics, and IoT backends.

Part 3

Storing data in the cloud

There is one guy named Singleton in your office who knows all about your file server. If Singleton is out of office, no one else can maintain the file server. As you can imagine, while Singleton is on vacation, the file server crashes. No one else knows where the backup is located, but the boss needs a document now or the company will lose a lot of money. If Singleton had stored his knowledge in a database, coworkers could look up the information. But because the knowledge and Singleton are tidily coupled, the information is unavailable.

Imagine a virtual machine where important files are located on hard disk. As long as the virtual machine is up and running, everything is fine. But everything fails all the time, including virtual machines. If a user uploads a document on your website, where is it stored? Chances are high that the document is persisted to hard disk on the virtual machine. Let's imagine that the document was uploaded to your website but persisted as an object in an independent object store. If the virtual machine fails, the document will still be available. If you need two virtual machines to handle the load on your website, they both have access to that document because it is not tightly coupled with a single virtual machine. If you separate your state from your virtual machine, you will be able to become fault tolerant and elastic. Let highly specialized solutions like object stores and databases persist your state.

AWS offers many ways to store your data. The following table can help you decide which service to use for your data at a high level. The comparison is only a rough overview. We recommend that you choose two or three services that best fit your use case and then jump into the details by reading the chapters to make your decision.

Table 1 Overview of data storage services

Service	Access	Maximum storage volume	Latency	Storage cost
S3	AWS API (SDKs, CLI), third-party tools	Unlimited	High	Very low
EBS (SSD)	Attached to an EC2 instance via network	16 TiB	Low	Low
EC2 Instance Store (SSD)	Attached to an EC2 instance directly	305 TB	Very low	Very low
EFS	NFSv4.1, for example, from an EC2 instance or on-premises	Unlimited	Medium	Medium
RDS (MySQL, SSD)	SQL	64 TiB	Medium	Low
ElastiCache	Redis/Memcached protocol	635 GiB	Low	High
DynamoDB	AWS API (SDKs, CLI)	Unlimited	Medium	Medium

Chapter 7 will introduce S3, a service offering object storage. You will learn how to integrate the object storage into your applications to implement a stateless server.

Chapter 8 is about block-level storage for virtual machines offered by AWS. You will learn how to operate legacy software on block-level storage.

Chapter 9 covers highly available block-level storage that can be shared across multiple virtual machines offered by AWS.

Chapter 10 introduces RDS, a service that offers managed relational database systems like PostgreSQL, MySQL, Oracle, or Microsoft SQL Server. If your applications use such a relational database system, this is an easy way to implement a stateless server architecture.

Chapter 11 introduces ElastiCache, a service that offers managed in-memory database systems like Redis or Memcached. If your applications need to cache data, you can use an in-memory database to externalize ephemeral state.

Chapter 12 will introduce DynamoDB, a service that offers a NoSQL database. You can integrate this NoSQL database into your applications to implement a stateless server.

7 Storing your objects: S3

This chapter covers

- Transferring files to S3 using the terminal
- Integrating S3 into your applications with SDKs
- Archiving data at low costs
- Hosting a static website with S3
- Diving into the internals of the S3 object store

Storing data comes with two challenges: dealing with ever-increasing volumes of data and ensuring durability. Solving the challenges is hard, or even impossible, if using disks connected to a single machine. For this reason, this chapter covers a revolutionary approach: a distributed data store consisting of a large number of machines connected over a network. This way, you can store nearly unlimited amounts of data by adding additional machines to the distributed data store. And because your data is always stored on more than one machine, you dramatically reduce the risk of losing that data.

You will learn about how to store images, videos, documents, executables, or any other kind of data on Amazon S3 in this chapter. Amazon S3 is a simple-to-use, fully managed distributed data store provided by AWS. Data is managed as objects, so the storage system is called an *object store*. We will show you how to use S3 to

back up your data, how to archive data at low cost, and how to integrate S3 into your own application for storing user-generated content, as well as how to host static websites on S3.

> **Not all examples are covered by the Free Tier**
>
> The examples in this chapter are not all covered by the Free Tier. A warning message appears when an example incurs costs. As for the other examples, as long as you follow the instructions and don't run them longer than a few days, you won't pay anything.

7.1 What is an object store?

Back in the old days, data was managed in a hierarchy consisting of folders and files. The file was the representation of the data. In an *object store*, data is stored as objects. Each object consists of a globally unique identifier, some metadata, and the data itself, as figure 7.1 illustrates. An object's *globally unique identifier* (GUID) is also known as its *key*; you can address the object from different devices and machines in a distributed system using the GUID.

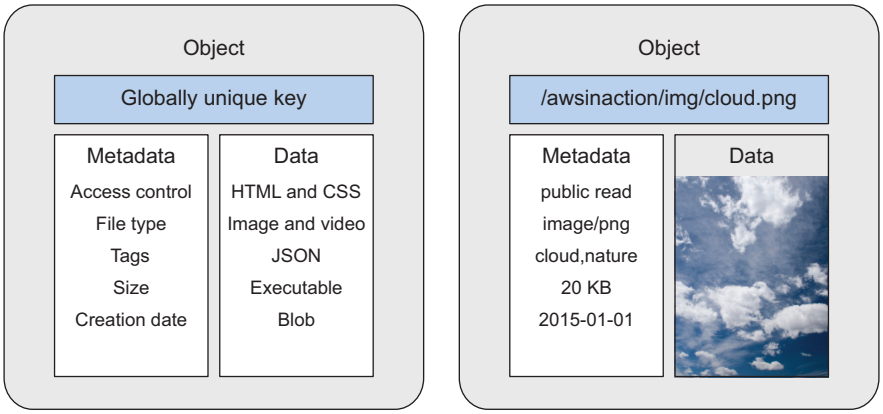


Figure 7.1 Objects stored in an object store have three parts: a unique ID, metadata describing the content, and the content itself (such as an image).

Typical examples for object metadata are:

- Date of last modification
- Object size
- Object's owner
- Object's content type

It is possible to request only an object's metadata without requesting the data itself. This is useful if you want to list objects and their metadata before accessing a specific object's data.

7.2 Amazon S3

Amazon S3 is a distributed data store, and one of the oldest services provided by AWS. *Amazon S3* is an acronym for *Amazon Simple Storage Service*. It's a typical web service that lets you store and retrieve data organized as objects via an API reachable over HTTPS. Here are some typical use cases:

- *Storing and delivering static website content*—For example, our blog https://cloudonaut.io is hosted on S3.
- *Backing up data*—For example, you can back up your photo library from your computer to S3 using the AWS CLI.
- *Storing structured data for analytics, also called a data lake*—For example, you can use S3 to store JSON files containing the results of performance benchmarks.
- *Storing and delivering user-generated content*—For example, we built a web application—with the help of the AWS SDK—that stores user uploads on S3.

Amazon S3 offers virtually unlimited storage space and stores your data in a highly available and durable way. You can store any kind of data, such as images, documents, and binaries, as long as the size of a single object doesn't exceed 5 TB. You have to pay for every GB you store in S3, and you also incur costs for every request and for all transferred data. As figure 7.2 shows, you can access S3 via the internet using HTTPS to upload and download objects. To access S3, you can use the Management Console, the CLI, SDKs, or third-party tools.

Figure 7.2 Uploading and downloading an object to S3 via HTTPS

S3 uses *buckets* to group objects. A bucket is a container for objects. It is up to you to create multiple buckets, each of which has a globally unique name, to separate data for different scenarios. By *unique*, we really mean unique—you have to choose a bucket name that isn't used by any other AWS customer in any other region. Figure 7.3 shows the concept.

You will learn how to upload and download data to S3 using the AWS CLI next.

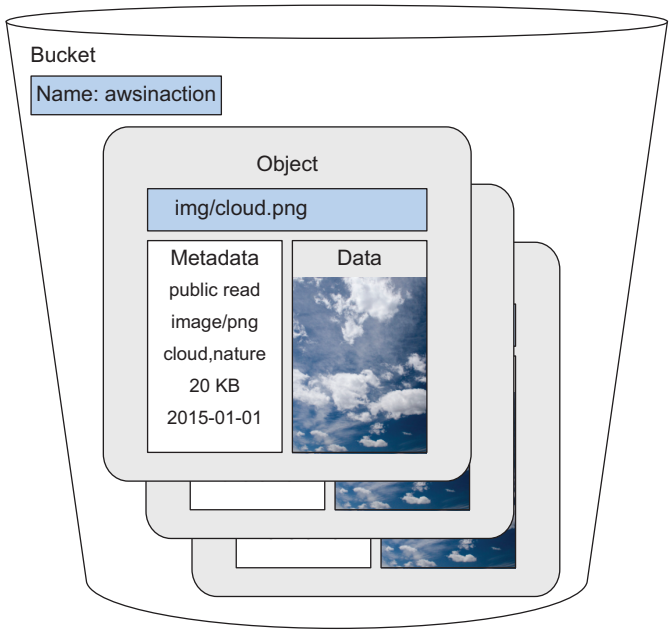


Figure 7.3 S3 uses buckets with globally unique names to group objects.

7.3 Backing up your data on S3 with AWS CLI

Critical data needs to be backed up to avoid loss. Backing up data at an offsite location decreases the risk of losing data, even during extreme conditions like natural disaster. But where should you store your backups? S3 allows you to store any data in the form of objects. The AWS object store is a perfect fit for your backup, allowing you to choose a location for your data as well as storing any amount of data with a pay-per-use pricing model.

In this section, you'll learn how to use the AWS CLI to upload data to and download data from S3. This approach isn't limited to offsite backups; you can use it in many other scenarios as well, such as the following:

- Sharing files with your coworkers or partners, especially when working from different locations
- Storing and retrieving artifacts needed to provision your virtual machines (such as application binaries, libraries, or configuration files)
- Outsourcing storage capacity to lighten the burden on local storage systems—in particular, for data that is accessed infrequently

First, you need to create a bucket for your data on S3. As we mentioned earlier, the name of the bucket must be unique among all other S3 buckets, even those in other regions and those of other AWS customers. To find a unique bucket name, it's useful to use a prefix or suffix that includes your company's name or your own name. Run the following command in the terminal, replacing `$yourname` with your name:

```
$ aws s3 mb s3://awsinaction-$yourname
```

Your command should look similar to this one:

```
$ aws s3 mb s3://awsinaction-awittig
```

In the unlikely event that you or another AWS customer has already created a bucket with this name, you will see the following error:

```
[... ] An error occurred (BucketAlreadyExists) [...]
```

In this case, you'll need to use a different value for `$yourname`.

Everything is ready for you to upload your data. Choose a folder you'd like to back up, such as your Desktop folder. Try to choose a folder with a total size less than 1 GB, to avoid long waiting times and exceeding the Free Tier. The following command uploads the data from your local folder to your S3 bucket. Replace `$path` with the path to your folder and `$yourname` with your name. `sync` compares your folder with the /backup folder in your S3 bucket and uploads only new or changed files:

```
$ aws s3 sync $path s3://awsinaction-$yourname/backup
```

Your command should look similar to this one:

```
$ aws s3 sync /Users/andreas/Desktop s3://awsinaction-awittig/backup
```

Depending on the size of your folder and the speed of your internet connection, the upload can take some time.

After uploading your folder to your S3 bucket to back it up, you can test the restore process. Execute the following command in your terminal, replacing `$path` with a folder you'd like to use for the restore (don't use the folder you backed up) and `$yourname` with your name. Your Downloads folder would be a good place to test the restore process:

```
$ aws s3 cp --recursive s3://awsinaction-$yourname/backup $path
```

Your command should look similar to this one:

```
$ aws s3 cp --recursive s3://awsinaction-awittig/backup/ \
➥ /Users/andreas/Downloads/restore
```

Again, depending on the size of your folder and the bandwidth of your internet connection, the download may take a while.

Versioning for objects

By default, S3 versioning is disabled for every bucket. Suppose you use the following steps to upload two objects:

1. Add an object with key A and data 1.
2. Add an object with key A and data 2.

(continued)

If you download the object with key A, you'll download data 2. The old data 1 doesn't exist any more.

You can change this behavior by turning on versioning for a bucket. The following command activates versioning for your bucket. Don't forget to replace `$yourname`:

```
$ aws s3api put-bucket-versioning --bucket awsinaction-$yourname \
➥ --versioning-configuration Status=Enabled
```

If you repeat the previous steps, the first version of object A consisting of data 1 will be accessible even after you add an object with key A and data 2. The following command retrieves all objects and versions:

```
$ aws s3api list-object-versions --bucket awsinaction-$yourname
```

You can now download all versions of an object.

Versioning can be useful for backing up and archiving scenarios. Keep in mind that the size of the bucket you'll have to pay for will grow with every new version.

You no longer need to worry about losing data. S3 is designed for 99.999999999% durability of objects over a year. For instance, when storing 100,000,000,000 objects on S3, you will lose only a single object per year on average.

After you've successfully restored your data from the S3 bucket, it's time to clean up. Execute the following command to remove the S3 bucket containing all the objects from your backup. You'll have to replace `$yourname` with your name to select the right bucket. `rb` removes the bucket; the `force` option deletes every object in the bucket before deleting the bucket itself:

```
$ aws s3 rb --force s3://awsinaction-$yourname
```

Your command should look similar to this one:

```
$ aws s3 rb --force s3://awsinaction-awittig
```

You're finished—you've uploaded and downloaded files to S3 with the help of the CLI.

Removing a bucket causes a `BucketNotEmpty` error

If you turn on versioning for your bucket, removing the bucket will cause a `BucketNotEmpty` error. Use the Management Console to delete the bucket in this case as follows:

1. Open the Management Console with your browser.
2. Go to the S3 service using the main navigation menu.
3. Select the bucket you want to delete.

4 Click the Empty button, and confirm permanently deleting all objects.
5 Wait until objects and versions have been deleted, and click the Exit button.
6 Select the bucket you want to delete.
7 Click the Delete button, and confirm deleting the bucket.

7.4 Archiving objects to optimize costs

In the previous section, you learned about backing up your data to S3. Storing 1 TB of data on S3 costs about $23 per month. Wouldn't it be nice to reduce the costs for storing data by 95%? Besides, by default, S3 comes with storage classes designed to archive data for long time spans.

Table 7.1 compares the storage class *S3 Standard* with storage classes intended for data archival.

Table 7.1 Differences between storing data with S3 and Glacier

	S3 Standard	S3 Glacier Instant Retrieval	S3 Glacier Flexible Retrieval	S3 Glacier Deep Archive
Storage costs for 1 GB per month in US East (N. Virginia)	$0.023	$0.004	$0.0036	$0.00099
Costs for 1,000 write requests	$0.005	$0.02	$0.03	$0.05
Costs for retrieving data	Low	High	High	Very High
Accessibility	Milliseconds	Milliseconds	1–5 minutes/ 3–5 hours/ 5–12 hours	12 hours/ 48 hours
Durability objective	99.999999999%	99.999999999%	99.999999999%	99.999999999%
Availability objective	99.99%	99.9%	99.99%	99.99%

The potential savings for storage costs are enormous. So what's the catch?

First, accessing data stored on S3 by using the storage classes S3 Glacier Instant Retrieval, S3 Glacier Flexible Retrieval, and S3 Glacier Deep Archive is expensive. Let's assume, you are storing 1 TB of data on S3 and decided to use storage type S3 Glacier Deep Archive. It will cost you about $120 to restore 1 TB of data stored in 1,000 files.

Second, fetching data from S3 Glacier Flexible Retrieval and S3 Glacier Deep Archive takes something between 1 minute and 48 hours, depending on the storage class and retrieval option.

Using the following example, we would like to explain what it means not to be able to access archived data immediately. Let's say you want to archive a document

for five years. You do not expect to access the document more than five times during this period.

> **Example not covered by Free Tier**
> The following example is not covered by the Free Tier. Archiving and restoring data as shown in the example will cost you less than $1. You will find information on how to delete all resources at the end of the section. Therefore, we recommend completing this section within a few days

Start by creating an S3 bucket that you will use to archive documents, as shown next. Replace `$yourname` with your name to get a unique bucket name:

```
$ aws s3 mb s3://awsinaction-archive-$yourname
```

Next, copy a document from your local machine to S3. The `--storage-class` parameter overrides the default storage class with `GLACIER`, which maps to the S3 Glacier Flexible Retrieval storage class. Replace `$path` with the path to a document, and `$yourname` with your name. Note the key of the object:

```
$ aws s3 cp --storage-class GLACIER $path \
➥ s3://awsinaction-archive-$yourname/
```

For instance, I run the following command:

```
$ aws s3 cp --storage-class GLACIER \
➥ /Users/andreas/Desktop/taxstatement-2022-07-01.pdf \
➥ s3://awsinaction-archive-awittig/
```

The key point is that you can't download the object. Replace `$objectkey` with the object's key that you noted down after uploading the document, and `$path` with the Downloads folder on your local machine:

```
$ aws s3 cp s3://awsinaction-archive-$yourname/$objectkey $path
```

For example, I'm getting the following error when trying to download my document taxstatement-2022-07-01.pdf:

```
$ aws s3 cp s3://awsinaction-archive-awittig/taxstatement-2022-07-01.pdf \
➥ ~/Downloads
warning: Skipping file s3://awsinaction-archive-awittig/
➥ taxstatement-2022-07-01.pdf. Object is of storage class GLACIER.
➥ Unable to perform download operations on GLACIER objects. You must
➥ restore the object to be able to perform the operation.
```

As mentioned in the error message, you need to restore the object before downloading it. By default, doing so will take three to five hours. That's why we will pay a little

extra—just a few cents—for expedited retrieval. Execute the following command after replacing `$yourname` with your name, and `$objectkey` with the object's key:

```
$ aws s3api restore-object --bucket awsinaction-archive-$yourname \
➥ --key $objectkey \
➥ --restore-request Days=1,,GlacierJobParameters={"Tier"="Expedited"}
```

This results in the following command in my scenario:

```
$ aws s3api restore-object --bucket awsinaction-archive-awittig \
➥ --key taxstatement-2022-07-01.pdf
➥ --restore-request Days=1,,GlacierJobParameters={"Tier"="Expedited"}
```

As you are using expedited retrieval, you need to wait one to five minutes for the object to become available for download. Use the following command to check the status of the object and its retrieval. Don't forget to replace `$yourname` with your name, and `$objectkey` with the object's key:

```
$ aws s3api head-object --bucket awsinaction-archive-$yourname \
➥ --key $objectkey
{
    "AcceptRanges": "bytes",
    "Expiration": "expiry-date=\"Wed, 12 Jul 2023 ...\", rule-id=\"...\"",
    "Restore": "ongoing-request=\"true\"",
    "LastModified": "2022-07-11T09:26:12+00:00",
    "ContentLength": 112,
    "ETag": "\"c25fa1df1968993d8e647c9dcd352d39\"",
    "ContentType": "binary/octet-stream",
    "Metadata": {},
    "StorageClass": "GLACIER"
}
```

Restoration of the object is still ongoing.

Repeat fetching the status of the object until `ongoing-request` flips to `false`:

```
{
    "AcceptRanges": "bytes",
    "Expiration": "expiry-date=\"Wed, 12 Jul 2023 ...\", rule-id=\"...\"",
    "Restore": "ongoing-request=\"false\", expiry-date=\"...\"",
    "LastModified": "2022-07-11T09:26:12+00:00",
    "ContentLength": 112,
    "ETag": "\"c25fa1df1968993d8e647c9dcd352d39\"",
    "ContentType": "binary/octet-stream",
    "Metadata": {},
    "StorageClass": "GLACIER"
}
```

The restoration is finished, with no ongoing restore requests.

After restoring the object, you are now able to download the document using the next code snippet. Replace `$objectkey` with the object's key that you noted down after uploading the document, and `$path` with the Downloads folder on your local machine:

```
$ aws s3 cp s3://awsinaction-archive-$yourname/$objectkey $path
```

In summary, the Glacier storage types are intended for archiving data that you need to access seldom, which means every few months or years. For example, we are using the S3 Glacier Deep Archive to store a remote backup of our MacBooks. Because we store another backup of our data on an external hard drive, the chances that we need to restore data from S3 are very low.

Cleaning up

Execute the following command to remove the S3 bucket containing all the objects from your backup. You'll have to replace `$yourname` with your name to select the right bucket. `rb` removes the bucket; the `force` option deletes every object in the bucket before deleting the bucket itself:

```
$ aws s3 rb --force s3://awsinaction-archive-$yourname
```

You've learned how to use S3 with the help of the CLI. We'll show you how to integrate S3 into your applications with the help of SDKs in the next section.

7.5 Storing objects programmatically

S3 is accessible using an API via HTTPS. This enables you to integrate S3 into your applications by making requests to the API programmatically. Doing so allows your applications to benefit from a scalable and highly available data store. AWS offers free SDKs for common programming languages like C++, Go, Java, JavaScript, .NET, PHP, Python, and Ruby. You can execute the following operations using an SDK directly from your application:

- Listing buckets and their objects
- Creating, removing, updating, and deleting (CRUD) objects and buckets
- Managing access to objects

Here are examples of how you can integrate S3 into your application:

- *Allow a user to upload a profile picture.* Store the image in S3, and make it publicly accessible. Integrate the image into your website via HTTPS.
- *Generate monthly reports (such as PDFs), and make them accessible to users.* Create the documents and upload them to S3. If users want to download documents, fetch them from S3.
- *Share data between applications.* You can access documents from different applications. For example, application A can write an object with the latest information about sales, and application B can download the document and analyze the data.

Integrating S3 into an application is one way to implement the concept of a *stateless server*. We'll show you how to integrate S3 into your application by diving into a simple web application called Simple S3 Gallery next. This web application is built on top of Node.js and uses the AWS SDK for JavaScript and Node.js. You can easily transfer what you learn from this example to SDKs for other programming languages; the concepts are the same.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify that everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples like the Simple S3 Gallery.

Do you want to get started with Node.js? We recommend *Node.js in Action* (second edition) by Alex Young et al. (Manning, 2017), or the video course *Node.js in Motion* by PJ Evans (Manning, 2018).

Next, we will dive into a simple web application called the Simple S3 Gallery. The gallery allows you to upload images to S3 and displays all the images you've already uploaded. Figure 7.4 shows Simple S3 Gallery in action. Let's set up S3 to start your own gallery.

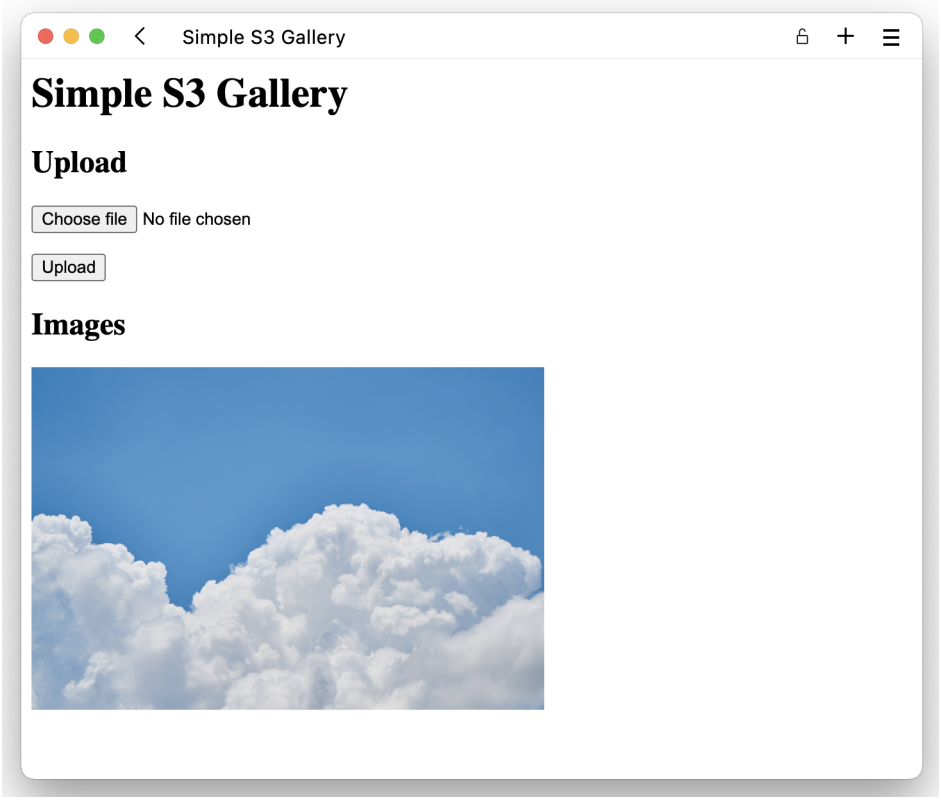


Figure 7.4 The Simple S3 Gallery app lets you upload images to an S3 bucket and then download them from the bucket for display.

7.5.1 Setting up an S3 bucket

To begin, you need to set up an empty bucket. Execute the following command, replacing `$yourname` with your name:

```
$ aws s3 mb s3://awsinaction-sdk-$yourname
```

Your bucket is now ready to go. Installing the web application is the next step.

7.5.2 Installing a web application that uses S3

You can find the Simple S3 Gallery application in /chapter07/gallery/ in the book's code folder. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

To start the web application, run the following command. Replace `$yourname` with your name; the name of the S3 bucket is then passed to the web application:

```
$ node server.js awsinaction-sdk-$yourname
```

> **Where is the code located?**
> You can find all the code in the book's code repository on GitHub: https://github.com/AWSinAction/code3. You can download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip.

After you start the server, you can open the gallery application. To do so, open http://localhost:8080 with your browser. Try uploading a few images.

7.5.3 Reviewing code access S3 with SDK

You've uploaded images to the Simple S3 Gallery and displayed images from S3. Inspecting parts of the code will help you understand how you can integrate S3 into your own applications. It's not a problem if you don't follow all the details of the programming language (JavaScript) and the Node.js platform; we just want you to get an idea of how to use S3 via SDKs.

UPLOADING AN IMAGE TO S3

You can upload an image to S3 with the SDK's `putObject()` function. Your application will connect to the S3 service and transfer the image via HTTPS. The next listing shows how to do so.

Listing 7.1 Uploading an image with the AWS SDK for S3

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

const s3 = new AWS.S3({
  'region': 'us-east-1'
});
```

Loads the AWS SDK

Instantiates the S3 client with additional configurations

```
const bucket = process.argv[2];

async function uploadImage(image, response) {
  try {
    await s3.putObject({
      Body: image,
      Bucket: bucket,
      Key: uuid.v4(),
      ACL: 'public-read',
      ContentLength: image.byteCount,
      ContentType: image.headers['content-type']
    }).promise();
    response.redirect('/');
  } catch (err) {
    console.error(err);
    response.status(500);
    response.send('Internal server error.');
  }
}
```

Image content

Uploads the image to S3

Name of the bucket

Allows everybody to read the image from bucket

Generates a unique key for the object

Size of image in bytes

Content type of the object (image/png)

Catching errors

Returns an error with the HTTP status code 500

The AWS SDK takes care of sending all the necessary HTTPS requests to the S3 API in the background.

Listing all the images in the S3 bucket

To display a list of images, the application needs to list all the objects in your bucket. This can be done with the S3 service's `listObjects()` function. The next code listing shows the implementation of the corresponding function in the server.js JavaScript file, acting as a web server.

Listing 7.2 Retrieving all the image locations from the S3 bucket

```
const bucket = process.argv[2];

async function listImages(response) {
  try {
    let data = await s3.listObjects({
      Bucket: bucket
    }).promise();
    let stream = mu.compileAndRender(
      'index.html',
      {
        Objects: data.Contents,
        Bucket: bucket
      }
    );
    stream.pipe(response);
  } catch (err) {
    console.error(err);
    response.status(500);
    response.send('Internal server error.');
  }
}
```

Reads the bucket name from the process arguments

The bucket name is the only required parameter.

Lists the objects stored in the bucket

Renders an HTML page based on the list of objects

Streams the response

Handles potential errors

Listing the objects returns the names of all the images from the bucket, but the list doesn't include the image content. During the uploading process, the access rights to the images are set to public read. This means anyone can download the images with the bucket name and a random key. The following listing shows an excerpt of the index.html template, which is rendered on request. The `Objects` variable contains all the objects from the bucket.

Listing 7.3 Template to render the data as HTML

```
[...]
<h2>Images</h2>
{{#Objects}}
  <p><img src=
➥ "https://s3.amazonaws.com/{{Bucket}}/{{Key}}"
➥ width="400px" ></p>
{{/Objects}}
[...]
```

Iterates over all objects

Puts together the URL to fetch an image from the bucket

You've now seen the three important parts of the Simple S3 Gallery integration with S3: uploading an image, listing all images, and downloading an image.

Cleaning up

Don't forget to clean up and delete the S3 bucket used in the example. Use the following command, replacing `$yourname` with your name:

```
$ aws s3 rb --force s3://awsinaction-sdk-$yourname
```

You've learned how to use S3 using the AWS SDK for JavaScript and Node.js. Using the AWS SDK for other programming languages is similar.

The next section is about a different scenario: you will learn how to host static websites on S3.

7.6 Using S3 for static web hosting

We started our blog https://cloudonaut.io in May 2015. The most popular blog posts, like "ECS vs. Fargate: What's the Difference?" (http://mng.bz/Xa2E), "Advanced AWS Networking: Pitfalls That You Should Avoid" (http://mng.bz/yaxe), and "CloudFormation vs. Terraform" (https://cloudonaut.io/cloudformation-vs-terraform/) have been read more than 200,000 times. But we did not need to operate any VMs to publish our blog posts. Instead, we used S3 to host our static website built with a static site generator, Hexo (https://hexo.io). This approach provides a cost-effective, scalable, and maintenance-free infrastructure for our blog.

You can host a static website with S3 and deliver static content like HTML, JavaScript, CSS, images (such as PNG and JPG), audio, and videos. Keep in mind, however, that you

can't execute server-side scripts like PHP or JSP. For example, it's not possible to host WordPress, a content management system based on PHP, on S3.

> **Increasing speed by using a CDN**
>
> Using a content-delivery network (CDN) helps reduce the load time for static web content. A CDN distributes static content like HTML, CSS, and images to nodes all around the world. If a user sends out a request for some static content, the request is answered from the nearest available node with the lowest latency. Various providers offer CDNs. Amazon CloudFront is the CDN offered by AWS. When using CloudFront, users connect to CloudFront to access your content, which is fetched from S3 or other sources. See the CloudFront documentation at http://mng.bz/M0m8 if you want to set this up; we won't cover it in this book.

In addition, S3 offers the following features for hosting a static website:

- *Defining a custom index document and error documents*—For example, you can define index.html as the default index document.
- *Defining redirects for all or specific requests*—For example, you can forward all requests from /img/old.png to /img/new.png.
- *Setting up a custom domain for an S3 bucket*—For example, Andreas might want to set up a domain like mybucket.andreaswittig.info pointing to his bucket.

7.6.1 Creating a bucket and uploading a static website

First you need to create a new S3 bucket. To do so, open your terminal and execute the following command, replacing `$BucketName` with your own bucket name. As we've mentioned, the bucket name has to be globally unique. If you want to link your domain name to S3, you must use your entire domain name as the bucket name:

```
$ aws s3 mb s3://$BucketName
```

The bucket is empty; you'll place an HTML document in it next. We've prepared a placeholder HTML file. Download it to your local machine from the following URL: http://mng.bz/aPyX. You can now upload the file to S3. Execute the following command to do so, replacing `$pathToPlacerholder` with the path to the HTML file you downloaded in the previous step and `$BucketName` with the name of your bucket:

```
$ aws s3 cp $pathToPlaceholder/helloworld.html \
➥ s3://$BucketName/helloworld.html
```

You've now created a bucket and uploaded an HTML document called helloworld .html. You need to configure the bucket next.

7.6.2 Configuring a bucket for static web hosting

By default, only you, the owner, can access files from your S3 bucket. Because you want to use S3 to deliver your static website, you'll need to allow everyone to view or download the documents included in your bucket. A *bucket policy* helps you control access to bucket objects globally. You already know from chapter 5 that policies are defined in JSON and contain one or more statements that either allow or deny specific actions on specific resources. Bucket policies are similar to IAM policies.

Download our bucket policy from the following URL: http://mng.bz/gROG. You need to edit the bucketpolicy.json file next, as shown in the following listing. Open the file with the editor of your choice, and replace `$BucketName` with the name of your bucket.

Listing 7.4 Bucket policy allowing read-only access to every object in a bucket

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

Allows access…

…for anyone

Reads objects

Your bucket

You can add a bucket policy to your bucket with the following command. Replace `$BucketName` with the name of your bucket and `$pathToPolicy` with the path to the bucketpolicy.json file:

```
$ aws s3api put-bucket-policy --bucket $BucketName \
➥ --policy file://$pathToPolicy/bucketpolicy.json
```

Every object in the bucket can now be downloaded by anyone. You need to enable and configure the static web-hosting feature of S3 next. To do so, execute the following command, replacing `$BucketName` with the name of your bucket:

```
$ aws s3 website s3://$BucketName --index-document helloworld.html
```

Your bucket is now configured to deliver a static website. The HTML document helloworld.html is used as index page. You'll learn how to access your website next.

7.6.3 Accessing a website hosted on S3

You can now access your static website with a browser. To do so, you need to choose the right endpoint. The endpoints for S3 static web hosting depend on your bucket's region. For us-east-1 (US East N. Virginia), the website endpoint looks like this:

```
http://$BucketName.s3-website-us-east-1.amazonaws.com
```

Replace `$BucketName` with your bucket. So if your bucket is called `awesomebucket` and was created in the default region us-east-1, your bucket name would be:

```
http://awesomebucket.s3-website-us-east-1.amazonaws.com
```

Open this URL with your browser. You should be welcomed by a Hello World website.

Please note that for some regions, the website endpoint looks a little different. Check S3 endpoints and quotas at http://mng.bz/epeq for details.

Linking a custom domain to an S3 bucket

If you want to avoid hosting static content under a domain like awsinaction.s3-website-us-east-1.amazonaws.com, you can link a custom domain to an S3 bucket, such as `awsinaction.example.com`. All you have to do is to add a CNAME record for your domain, pointing to the bucket's S3 endpoint. The domain name system provided by AWS allowing you to create a CNAME record is called Route 53.

The CNAME record will work only if you comply with the following rules:

- *Your bucket name must match the CNAME record name.* For example, if you want to create a CNAME for awsinaction.example.com, your bucket name must be awsinaction.example.com as well.
- *CNAME records won't work for the primary domain name (such as example.com).* You need to use a subdomain for CNAMEs like awsinaction or www. If you want to link a primary domain name to an S3 bucket, you need to use the Route 53 DNS service from AWS.

Linking a custom domain to your S3 bucket works only for HTTP. If you want to use HTTPS (and you probably should), use AWS CloudFront together with S3. AWS CloudFront accepts HTTPS from the client and forwards the request to S3.

Cleaning up

Don't forget to clean up your bucket after you finish the example. To do so, execute the following command, replacing `$BucketName` with the name of your bucket:

```
$ aws s3 rb --force s3://$BucketName
```

7.7 Protecting data from unauthorized access

Not a week goes by without a frightening announcement that an organization has leaked confidential data from Amazon S3 accidentally. Why is that?

While reading through this chapter, you have learned about different scenarios for using S3. For example, you used S3 to back up data from your local machine. Also, you hosted a static website on S3. So, S3 is used to store sensitive data as well as public data. This can be a dangerous mix because a misconfiguration might cause a data leak.

To mitigate the risk, we recommend you enable Block Public Access for all your buckets as illustrated in figure 7.5 and shown next. By doing so, you will disable public access to all the buckets belonging to your AWS account. This will break S3 website hosting or any other form of accessing S3 objects publicly.

1. Open the AWS Management Console and navigate to S3.
2. Select Block Public Access Settings for This Account from the subnavigation menu.
3. Enable Block All Public Access, and click the Save Changes button.

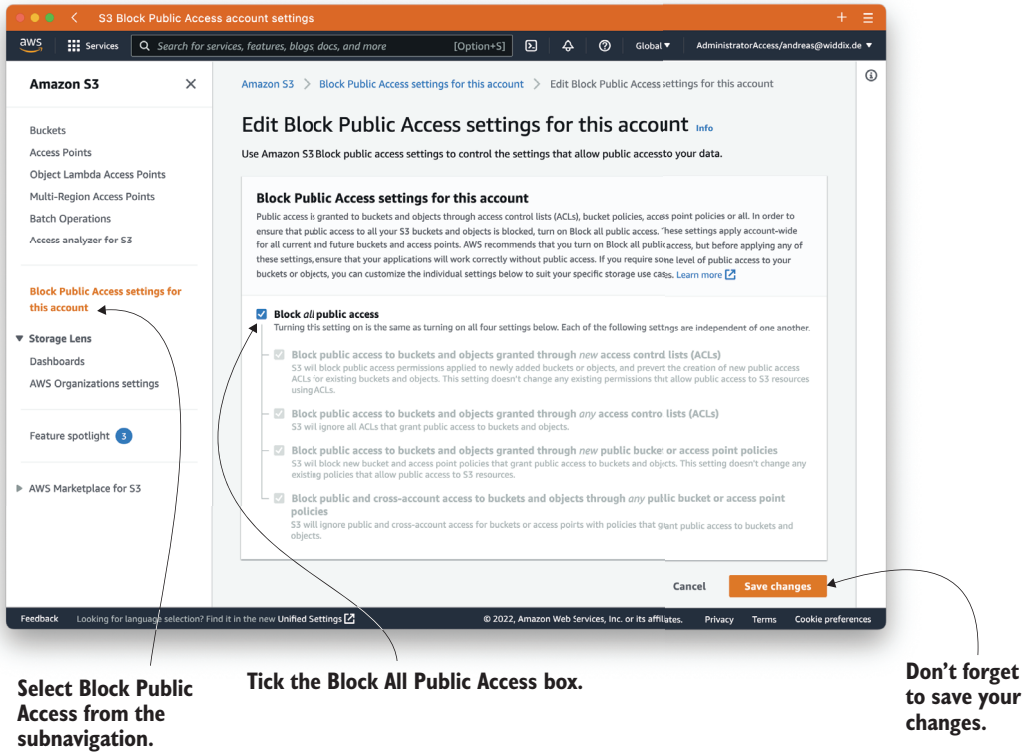


Figure 7.5 Enable Block Public Access for all S3 buckets to avoid data leaks.

In case you really need buckets with both sensitive data and public data, you should enable Block Public Access not on the account level but for all buckets with sensitive data individually instead.

Check out our blog post "How to Avoid S3 Data Leaks?" at https://cloudonaut.io/s3-security-best-practice/ if you are interested in further advice.

7.8 Optimizing performance

By default, S3 handles 3,500 writes and 5,500 reads per second. If your workload requires higher throughput, you need to consider the following when coming up with the naming scheme for the object keys.

Objects are stored without a hierarchy on S3. There is no such thing as a directory. All you do is specify an object key, as discussed at the beginning of the chapter. However, using a prefix allows you to structure the object keys.

By default, the slash character (/) is used as the prefix delimiter. So, `archive` is the prefix in the following example of object keys:

```
archive/image1.png
archive/image2.png
archive/image3.png
archive/image4.png
```

Be aware that the maximum throughput per partitioned prefix is 3,500 writes and 5,500 reads per second. Therefore, you cannot read more than 5,500 objects from the prefix `archive` per second.

To increase the maximum throughput, you need to distribute your objects among additional prefixes. For example, you could organize the objects from the previous example like this:

```
archive/2021/image1.png
archive/2021/image2.png
archive/2022/image3.png
archive/2022/image4.png
```

By doing so, you can double the maximum throughput when reading from archive/2021 and archive/2022 as illustrated in figure 7.6.

In summary, the structure of your object keys has an effect on the maximum read and write throughput.

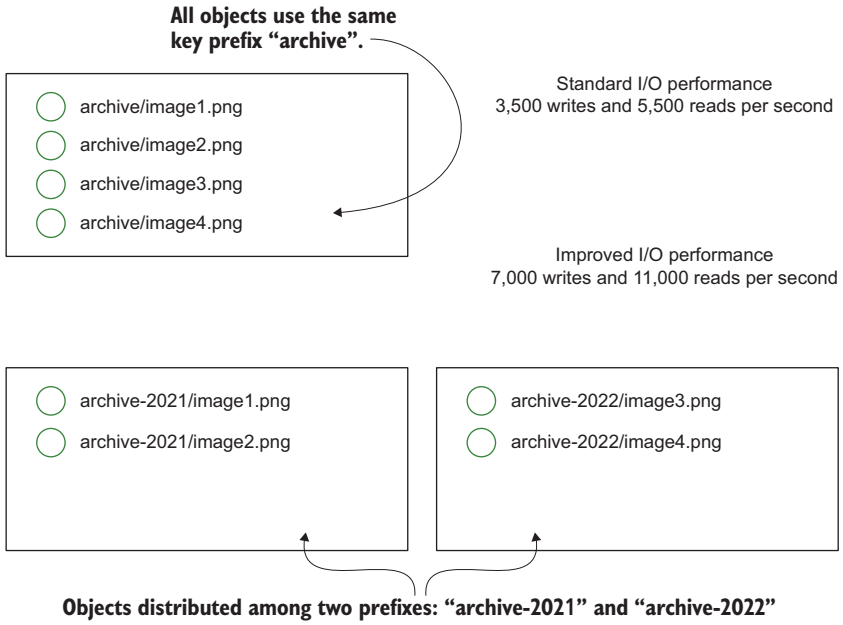


Figure 7.6 To improve I/O performance, distribute requests among multiple object key prefixes.

Summary

- An object consists of a unique identifier, metadata to describe and manage the object, and the content itself. You can save images, documents, executables, or any other content as an object in an object store.
- Amazon S3 provides endless storage capacity with a pay-per-use model. You are charged for storage as well as read and write requests.
- Amazon S3 is an object store accessible only via HTTP(S). You can upload, manage, and download objects with the CLI, SDKs, or the Management Console. The storage classes Glacier Instant Retrieval, Glacier Flexible Retrieval, and Glacier Deep Archive are designed to archive data at low cost.
- Integrating S3 into your applications will help you implement the concept of a stateless server, because you don't have to store objects locally on the server.
- Enable Block Public Access for all buckets, or at least those buckets that contain sensitive information to avoid data leaks.
- When optimizing for high performance, make sure to use many different key prefixes instead of similar ones or the same prefix for all objects.

8 Storing data on hard drives: EBS and instance store

This chapter covers

- Attaching persistent storage volumes to an EC2 instance
- Using temporary storage attached to the host system
- Backing up volumes
- Testing and tweaking volume performance
- Differences between persistent (EBS) and temporary volumes (instance store)

Imagine your task is to migrate an enterprise application being hosted on-premises to AWS. Typically, legacy applications read and write files from a filesystem. Switching to object storage, as described in the previous chapter, is not always possible or easy. Fortunately, AWS offers good old block-level storage as well, allowing you to migrate your legacy application without the need for expensive modifications.

Block-level storage with a disk filesystem (FAT32, NTFS, ext3, ext4, XFS, and so on) can be used to store files as you would on a personal computer. A *block* is a sequence of bytes and the smallest addressable unit. The OS is the intermediary

between the application that needs to access files and the underlying filesystem and block-level storage. The disk filesystem manages where (at what block address) your files are stored. You can use block-level storage only in combination with an EC2 instance where the OS is running.

The OS provides access to block-level storage via open, write, and read system calls. The simplified flow of a read request goes like this:

1. An application wants to read the file /path/to/file.txt and makes a read system call.
2. The OS forwards the read request to the filesystem.
3. The filesystem translates /path/to/file.txt to the block on the disk where the data is stored.

Applications like databases that read or write files by using system calls must have access to block-level storage for persistence. You can't tell a MySQL database to store its files in an object store because MySQL uses system calls to access files.

> **Not all examples are covered by the Free Tier**
>
> The examples in this chapter are not all covered by the Free Tier. A warning message appears when an example incurs costs. Nevertheless, as long as you don't run all other examples longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

AWS provides two kinds of block-level storage:

- *A* persistent block-level storage *volume connected via network*—This is the best choice for most problems, because it is independent of your virtual machine's life cycle and replicates data among multiple disks automatically to increase durability and availability.
- *A* temporary block-level storage *volume physically attached to the host system of the virtual machine*—This is interesting if you're optimizing for performance, because it is directly attached to the host system and, therefore, offers low latency and high throughput when accessing your data.

The next three sections will introduce and compare these two solutions by connecting storage with an EC2 instance, doing performance tests, and exploring how to back up the data.

8.1 Elastic Block Store (EBS): Persistent block-level storage attached over the network

Elastic Block Store (EBS) provides persistent block-level storage with built-in data replication. Typically, EBS is used in the following scenarios:

- Operating a relational database system on a virtual machine
- Running a (legacy) application that requires a filesystem to store data on EC2
- Storing and booting the operating system of a virtual machine

An EBS volume is separate from an EC2 instance and connected over the network, as shown in figure 8.1. EBS volumes have the following characteristics:

- They aren't part of your EC2 instances; they're attached to your EC2 instance via a network connection. If you terminate your EC2 instance, the EBS volumes remain.
- They are either not attached to an EC2 instance or attached to exactly one EC2 instance at a time.
- They can be used like typical hard disks.
- They replicate your data on multiple disks to prevent data loss due to hardware failures.

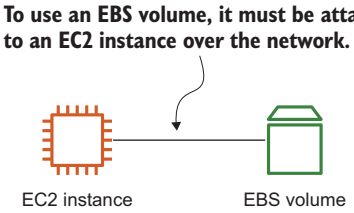


Figure 8.1 EBS volumes are independent resources but can be used only when attached to an EC2 instance.

EBS volumes have one big advantage: they are not part of the EC2 instance; they are an independent resource. No matter whether you stop your virtual machine or your virtual machine fails because of a hardware defect, your volume and your data will remain.

By default, AWS sets the `DeleteOnTermination` attribute to `true` for the root volume of each EC2 instance. This means, whenever you terminate the EC2 instance, the EBS volume acting as the root volume gets deleted automatically. In contrast, AWS sets the `DeleteOnTermination` attribute to `false` for all other EBS volumes attached to an EC2 instance. When you terminate the EC2 instance, those EBS volumes will remain. If you need to modify the default behavior, it is possible to override the initial value for the `DeleteOnTermination` attribute.

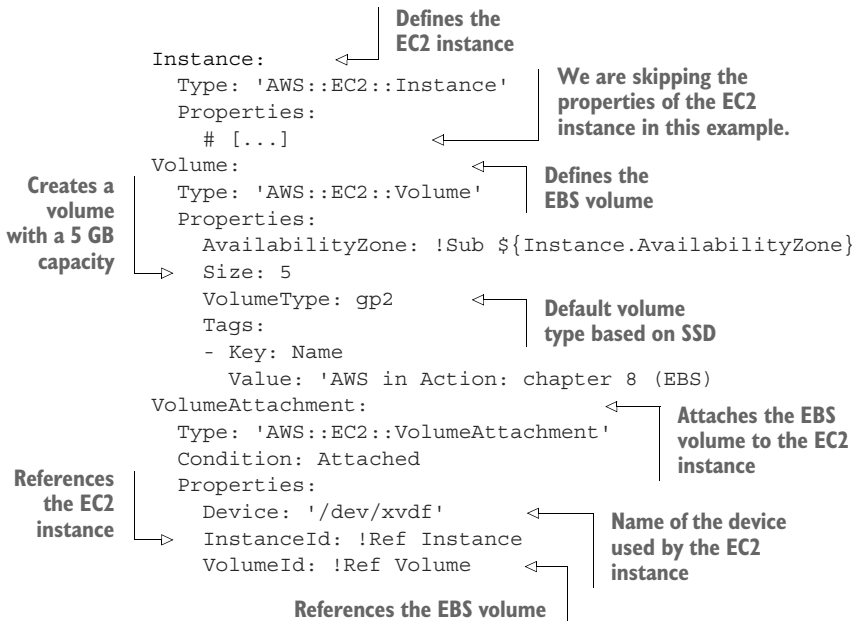
WARNING You can't easily attach the same EBS volume to multiple virtual machines! If you are still interested in attaching an EBS volume to many EC2 instances, read http://mng.bz/p68w carefully and see if the many limitations

still support your workload. See chapter 9 if you are looking for a network filesystem.

8.1.1 Creating an EBS volume and attaching it to your EC2 instance

Let's return to the example from the beginning of the chapter. You are migrating a legacy application to AWS. The application needs to access a filesystem to store data. Because the data contains business-critical information, durability and availability are important. Therefore, you create an EBS volume for persistent block storage. The legacy application runs on a virtual machine, and the volume is attached to the EC2 instance to enable access to the block-level storage.

The following bit of code demonstrates how to create an EBS volume and attach it to an EC2 instance with the help of CloudFormation:



An EBS volume is a standalone resource. This means your EBS volume can exist without an EC2 instance, but you need an EC2 instance to access the EBS volume.

8.1.2 Using EBS

To help you explore EBS, we've prepared a CloudFormation template located at https://s3.amazonaws.com/awsinaction-code3/chapter08/ebs.yaml. Create a stack based on that template by clicking the CloudFormation Quick-Create Link (http://mng.bz/O6la), select the default VPC and a random subnet, and set the `AttachVolume` parameter to `yes`. Don't forget to check the box marked "I Acknowledge That AWS CloudFormation Might Create IAM Resources." After creating the stack, use the SSM Session Manager to connect to the instance.

You can see the attached EBS volumes using `lsblk`. Usually, EBS volumes can be found somewhere in the range of /dev/xvdf to /dev/xvdp. The root volume (/dev/xvda) is an exception—it's based on the AMI you choose when you launch the EC2 instance and contains everything needed to boot the instance (your OS files), as shown here:

```
$ lsblk
NAME    MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvda    202:0    0   8G  0 disk
└─xvda1 202:1    0   8G  0 part /
xvdf    202:80   0   5G  0 disk
```

The root volume, an EBS volume with a size of 8 GiB

An additional volume, an EBS volume with a size of 5 GiB

The first time you use a newly created EBS volume, you must create a filesystem. You could also create partitions, but in this case, the volume size is only 5 GB, so you probably don't want to split it up further. Because you can create EBS volumes in any size and attach multiple volumes to your EC2 instance, partitioning a single EBS volume is uncommon. Instead, you should create volumes at the size you need (1 GB to 16 TB); if you need two separate scopes, create two volumes. In Linux, you can create a filesystem on the additional volume with the help of `mkfs`. The following example creates an XFS filesystem:

```
$ sudo mkfs -t xfs /dev/xvdf
meta-data=/dev/xvdf              isize=512    agcount=4, agsize=327680 blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=1        finobt=1, sparse=0
data     =                       bsize=4096   blocks=1310720, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=1
log      =internal log           bsize=4096   blocks=2560, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
```

After the filesystem has been created, you can mount the device as follows:

```
$ sudo mkdir /data
$ sudo mount /dev/xvdf /data
```

To see mounted volumes, use `df` like this:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        484M     0  484M   0% /dev
tmpfs           492M     0  492M   0% /dev/shm
tmpfs           492M  348K  491M   1% /run
tmpfs           492M     0  492M   0% /sys/fs/cgroup
/dev/xvda1      8.0G  1.5G  6.6G  19% /
/dev/xvdf       5.0G   38M  5.0G   1% /data
```

Root volume

Additional volume

EBS volumes are independent of your virtual machine. To see this in action, as shown in the next code snippet, you will save a file to a volume, unmount, and detach the

volume. Afterward, you will attach and mount the volume again. The data will still be available!

```
$ sudo touch /data/testfile
$ sudo umount /data
```

Creates testfile in /data

Unmounts the volume

Open the AWS Management Console and update the CloudFormation stack named `ebs`. Keep the current template, but update the `AttachVolume` parameter from `yes` to `no` and update the stack. This will detach the EBS volume from the EC2 instance. After the update is completed, only your root device is left, as shown here:

```
$ lsblk
NAME    MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvda    202:0    0   8G  0 disk
└─xvda1 202:1    0   8G  0 part /
```

The testfile in /data is also gone, illustrated next:

```
$ ls /data/testfile
ls: cannot access /data/testfile: No such file or directory
```

Next, attach the EBS volume again. Open the AWS Management Console and update the CloudFormation stack named `ebs`. Keep the current template but set the `AttachVolume` parameter to `yes`. Then, update the CloudFormation stack and wait for the changes to be applied. The volume `/dev/xvdf` is available again, as shown here:

```
$ sudo mount /dev/xvdf /data
$ ls /data/testfile
/data/testfile
```

Mounts the attached volume again

Checks whether testfile is still in /data

Voilà! The file testfile that you created in /data is still there.

8.1.3 Tweaking performance

Performance testing of hard disks is divided into read and write tests. To test the performance of your volumes, you will use a simple tool named `dd`, which can perform block-level reads and writes between a source `if=/path/to/source` and a destination `of=/path/to/destination`, shown next. For comparison, you'll run a performance test for a temporary block-level storage volume in the following section:

```
$ sudo dd if=/dev/zero of=/data/tempfile bs=1M count=1024 \
➥ conv=fdatasync,notrunc
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 15.8331 s, 67.8 MB/s

$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
```

Writes 1 MB 1,024 times

67.8 MB/s write performance

Flushes caches

```
$ sudo dd if=/data/tempfile of=/dev/null bs=1M
➥ count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 15.7875 s, 68.0 MB/s
```

Reads 1 MB 1,024 times

68.0 MB/s read performance

Keep in mind that performance can be different depending on your actual workload. This example assumes that the file size is 1 MB. If you're hosting websites, you'll most likely deal with lots of small files instead.

EBS performance is more complicated. Performance depends on the type of EC2 instance as well as the EBS volume type. EC2 instances with EBS optimization benefit by having dedicated bandwidth to their EBS volumes. Table 8.1 gives an overview of EC2 instance types that are EBS-optimized by default. Some older instance types can be optimized for an additional hourly charge whereas others do not support EBS optimization at all. Input/output operations per second (IOPS) are measured using a standard 16 KB I/O operation size.

Table 8.1 What performance can be expected from modern instance types? Your mileage may vary.

Use case	Instance type	Baseline bandwidth (Mbps)	Maximum bandwidth (Mbps)
General purpose	m6a.large–m6a.48xlarge	531–40,000	6,666–40,000
Compute optimized	c6g.medium–c6g.16xlarge	315– 9,000	4,750–19,000
Memory optimized	r6i.large–r6i.32xlarge	650–40,000	10,000–40,000
Memory and network optimized	x2idn.16xlarge–x2idn.32xlarge	40,000–80,000	40,000–80,000

WARNING Performance depends heavily on your workload: read versus write, as well as the size of your I/O operations (a bigger operation size equates to more throughput).

Depending on your storage workload, you must choose an EC2 instance that can deliver the bandwidth you require. Additionally, your EBS volume must be balanced with the amount of bandwidth. Table 8.2 shows the different EBS volume types and how they perform.

Table 8.2 How EBS volume types differ

Volume type	Volume size	MiB/s	IOPS	Performance burst	Price
General Purpose SSD (gp3)	1 GiB–16 TiB	1,000	3,000 per default, plus as much as you provision (up to 500 IOPS per GiB or 16,000 IOPS)	n/a	$$$$

Table 8.2 How EBS volume types differ *(continued)*

Volume type	Volume size	MiB/s	IOPS	Performance burst	Price
General Purpose SSD (gp2)	1 GiB–16 TiB	250	3 per GiB (up to 16,000)	3,000 IOPS	$$$$$
Provisioned IOPS SSD (io2 Block Express)	4 GiB–64 TiB	4000	As much as you provision (up to 500 IOPS per GiB or 256,000 IOPS)	n/a	$$$$$$
Provisioned IOPS SSD (io2)	4 GiB–16 TiB	1000	As much as you provision (up to 500 IOPS per GiB or 64,000 IOPS)	n/a	$$$$$$
Provisioned IOPS SSD (io1)	4 GiB–16 TiB	1000	As much as you provision (up to 50 IOPS per GiB or 64,000 IOPS)	n/a	$$$$$$$
Throughput Optimized HDD (st1)	125 GiB–16 TiB	40 per TiB (up to 500)	500	250 MiB/s per TiB (up to 500 MiB/s)	$$
Cold HDD (sc1)	125 GiB–16 TiB	12 per TiB (up to 250)	250	80 MiB/s per TiB (up to 250 MiB/s)	$
EBS Magnetic HDD (standard)	1 GiB–1 TiB	40-90	40-200 (100 on average)	Hundreds	$$$

Here are typical scenarios for the different volume types:

- Use General Purpose SSD (gp3) as the default for most workloads with medium load and a random access pattern. For example, use this as the boot volume or for all kinds of applications with low to medium I/O load.
- I/O-intensive workloads access small amounts of data randomly. Provisioned IOPS SSD (io2) offers throughput guarantees, for example, for large and business-critical database workloads.
- Use Throughput Optimized HDD (st1) for workloads with sequential I/O and huge amounts of data, such as Big Data workloads. Don't use this volume type for workloads in need of small and random I/O.
- Cold HDD (sc1) is a good fit when you are looking for a low-cost storage option for data you need to access infrequently and sequentially. Don't use this volume type for workloads in need of small and random I/O.
- EBS Magnetic HDD (standard) is an older volume type from a previous generation. It might be a good option when you need to access your data infrequently.

Gibibyte (GiB) and Tebibyte (TiB)

The terms gibibyte (GiB) and tebibyte (TiB) aren't used often; you're probably more familiar with gigabyte and terabyte. But AWS uses them in some places. Here's what they mean:

1 TiB = 2^{40} bytes = 1,099,511,627,776 bytes	1 TB = 10^{12} bytes = 1,000,000,000,000 bytes
1 GiB = 2^{30} bytes = 1,073,741,824 bytes	1 GB = 10^{9} bytes = 1,000,000,000 bytes

Or, in other words, 1 TiB is 1.0995 TB and 1 GiB is 1.074 GB.

EBS volumes are charged based on the size of the volume, no matter how much data you store in the volume. If you provision a 100 GiB volume, you pay for 100 GiB, even if you have no data on the volume. If you use EBS Magnetic HDD (standard) volumes, you must also pay for every I/O operation you perform. Provisioned IOPS SSD (io1) volumes are additionally charged based on the provisioned IOPS. Use the AWS Simple Monthly Calculator at https://calculator.aws/ to determine how much your storage setup will cost.

We advise you to use general-purpose (SSD) volumes as the default. If your workload requires more IOPS, then go with provisioned IOPS (SSD). You can attach multiple EBS volumes to a single EC2 instance to increase overall capacity or for additional performance.

8.1.4 Backing up your data with EBS snapshots

EBS volumes replicate data on multiple disks automatically and are designed for an annual failure rate (AFR) of 0.1% and 0.2%. This means on average you should expect to lose 0.5–1 of 500 volumes per year. To plan for an unlikely (but possible) failure of an EBS volume, or more likely a human failure, you should create backups of your volumes regularly. Fortunately, EBS offers an optimized, easy-to-use way to back up EBS volumes with EBS snapshots. A *snapshot* is a block-level incremental backup. If your volume is 5 GiB in size, and you use 1 GiB of data, your first snapshot will be around 1 GiB in size. After the first snapshot is created, only the changes will be persisted, to reduce the size of the backup. EBS snapshots are charged based on how many gigabytes you use.

You'll now create a snapshot using the CLI. Before you can do so, you need to know the EBS volume ID. You can find it as the `VolumeId` output of the CloudFormation stack, or by running the following:

```
$ aws ec2 describe-volumes \
➥ --filters "Name=size,Values=5" --query "Volumes[].VolumeId" \
➥ --output text
vol-043a5516bc104d9c6
```

The output shows the $VolumeId.

With the volume ID, you can go on to create a snapshot like this:

```
$ aws ec2 create-snapshot --volume-id $VolumeId
{
  "Description": "",
  "Encrypted": false,
  "OwnerId": "163732473262",
  "Progress": "",
  "SnapshotId": "snap-0babfe807decdb918",
  "StartTime": "2022-08-25T07:59:50.717000+00:00",
  "State": "pending",
  "VolumeId": "vol-043a5516bc104d9c6",
  "VolumeSize": 5,
  "Tags": []
}
```

Replace $VolumeId with the ID of your volume.

Note the ID of the snapshot: $SnapshotId.

EBS is still creating your snapshot.

Creating a snapshot can take some time, depending on how big your volume is and how many blocks have changed since the last backup. You can see the status of the snapshot by running the following:

```
$ aws ec2 describe-snapshots --snapshot-ids $SnapshotId
{
  "Snapshots": [
    {
      "Description": "",
      "Encrypted": false,
      "OwnerId": "163732473262",
      "Progress": "100%",
      "SnapshotId": "snap-0babfe807decdb918",
      "StartTime": "2022-08-25T07:59:50.717000+00:00",
      "State": "completed",
      "VolumeId": "vol-043a5516bc104d9c6",
      "VolumeSize": 5,
      "StorageTier": "standard"
    }
  ]
}
```

Replace $VolumeId with the ID of your snapshot.

Progress of your snapshot

The snapshot has reached the state completed.

Creating a snapshot of an attached, mounted volume is possible but can cause problems with writes that aren't flushed to disk. You should either detach the volume from your instance or stop the instance first. If you absolutely must create a snapshot while the volume is in use, you can do so safely as follows:

1. Freeze all writes by running `sudo fsfreeze -f /data` on the virtual machine.
2. Create a snapshot and wait until it reaches the pending state.
3. Unfreeze to resume writes by running `sudo fsfreeze -u /data` on the virtual machine.
4. Wait until the snapshot is completed.

Unfreeze the volume as soon as the snapshot reaches the state `pending` . You don't have to wait until the snapshot has finished.

With an EBS snapshot, you don't have to worry about losing data due to a failed EBS volume or human failure. You are able to restore your data from your EBS snapshot.

To restore a snapshot, you must create a new EBS volume based on that snapshot. Execute the following command in your terminal, replacing `$SnapshotId` with the ID of your snapshot:

```
$ aws ec2 create-volume --snapshot-id $SnapshotId \
➥ --availability-zone us-east-1a
{
    "AvailabilityZone": "us-east-1a",
    "CreateTime": "2022-08-25T08:08:49+00:00",
    "Encrypted": false,
    "Size": 5,
    "SnapshotId": "snap-0babfe807decdb918",
    "State": "creating",
    "VolumeId": "vol-0bf4fdf3816f968c5",
    "Iops": 100,
    "Tags": [],
    "VolumeType": "gp2",
    "MultiAttachEnabled": false
}
```

The ID of your snapshot used to create the volume

Choose data center.

The $RestoreVolumeId of the volume restored from your snapshot

Cleaning up

Don't forget to delete the snapshot, the volume, and your stack. The following code will delete the snapshot and volume. Don't forget to replace `$SnapshotId` with the ID of the EBS snapshot you created earlier and `$RestoreVolumeId` with the ID of the EBS volume you created by restoring the snapshot:

```
$ aws ec2 delete-snapshot --snapshot-id $SnapshotId
$ aws ec2 delete-volume --volume-id $RestoreVolumeId
```

Also delete your CloudFormation stack named `ebs` after you finish this section to clean up all used resources as follows:

```
$ aws cloudformation delete-stack --stack-name ebs
```

8.2 Instance store: Temporary block-level storage

An *instance store* provides block-level storage directly attached to the physical machine hosting the virtual machine. Figure 8.2 shows that the instance store is part of an EC2 instance and available only if your instance is running; it won't persist your data if you stop or terminate the instance. You don't pay separately for an instance store; instance store charges are included in the EC2 instance price.

In comparison to an EBS volume, which is connected to your EC2 instance over the network, the instance store depends upon the EC2 instance and can't exist without it. The instance store will be deleted when you stop or terminate the virtual machine.

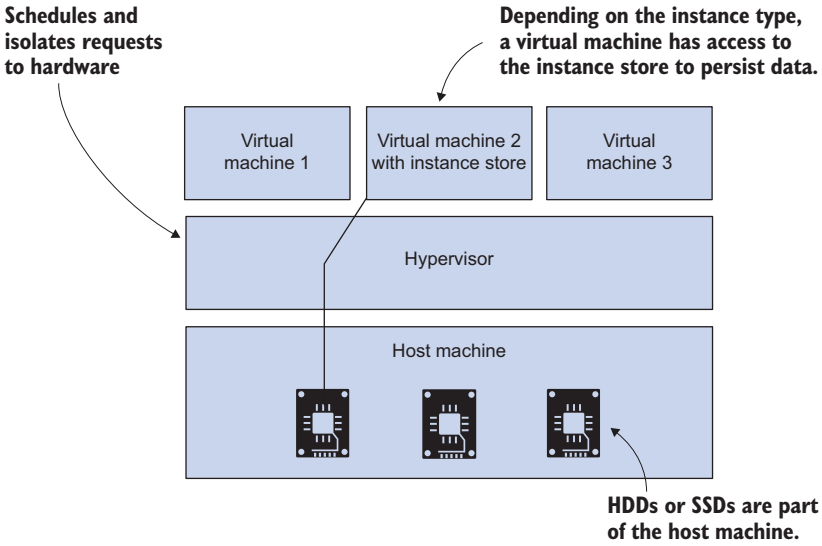


Figure 8.2 The instance store is part of your EC2 instance and uses the host machine's HDDs or SSDs.

Don't use an instance store for data that must not be lost; use it for temporary data only. For example, it is not a big deal to cache data on instance store, because you are able to restore the data from its origin anytime.

In rare cases, it is also advisable to persist data on instance storage, such as whenever the application is replicating data among multiple machines by default. For example, many NoSQL database systems use a cluster of machines to replicate data. The benefit of doing so is low latency and high throughput. But be warned, this is for experts in the field of distributed systems and storage only. If in doubt, use EBS instead.

WARNING If you stop or terminate your EC2 instance, the instance store is lost. *Lost* means all data is destroyed and can't be restored!

Note that most EC2 instance types do not come with instance storage. Only some instance families come with instance storage included. AWS offers SSD and HDD instance stores from 4 GB up to 335,520 GB. Table 8.3 shows a few EC2 instance families providing instance stores.

Table 8.3 Instance families with instance stores

Use case	Instance type	Instance store type	Instance store size in GB
General purpose	m6id.large m6id.32xlarge	SSD	118 7600
Compute optimized	c6id.large c6id.32xlarge	SSD	118 7600

Table 8.3 Instance families with instance stores *(continued)*

Use case	Instance type	Instance store type	Instance store size in GB
Memory optimized	r6id.large r6id.32xlarge	SSD	118 7600
Storage optimized	i4i.large i4i.32xlarge	SSD	468 30000
Storage optimized	d3.xlarge d3.8xlarge	HDD	5940 47520
Storage optimized	d3en.xlarge d3en.12xlarge	HDD	27960 335520

The following listing demonstrates how to use an instance store with CloudFormation. Instance stores aren't standalone resources like EBS volumes; the instance store is part of your EC2 instance.

Listing 8.1 Using an instance store with CloudFormation

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    IamInstanceProfile: 'ec2-ssm-core'
    ImageId: 'ami-061ac2e015473fbe2'
    InstanceType: 'm6id.large'
    SecurityGroupIds:
    - !Ref SecurityGroup
    SubnetId: !Ref Subnet
```

Chooses an instance type with an instance store

Read on to see how you can use the instance store.

8.2.1 Using an instance store

To help you explore instance stores, we created the CloudFormation template located at https://s3.amazonaws.com/awsinaction-code3/chapter08/instancestore.yaml. Create a CloudFormation stack based on the template by clicking the CloudFormation Quick-Create Link (http://mng.bz/YK5a).

> **WARNING** Starting a virtual machine with instance type `m6id.large` will incur charges. See https://aws.amazon.com/ec2/pricing/on-demand/ if you want to find out the current hourly price.

Create a stack based on that template, and select the default VPC and a random subnet. After creating the stack, use the SSM Session Manager to connect to the instance and explore the available devices, as shown in the following code snippet.

> **WARNING** You might run into the following error: "Your requested instance type (`m6id.large`) is not supported in your requested Availability Zone." Not

all regions/availability zones support `m6id.large` instance types. Select a different subnet and try again.

```
$ lsblk
NAME          MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
nvme1n1       259:0    0 109.9G  0 disk
nvme0n1       259:1    0     8G  0 disk
├─nvme0n1p1   259:2    0     8G  0 part /
└─nvme0n1p128 259:3    0     1M  0 part
```

The instance store device

The EBS root volume device

To see the mounted volumes, use this command:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        3.9G     0  3.9G   0% /dev
tmpfs           3.9G     0  3.9G   0% /dev/shm
tmpfs           3.9G  348K  3.9G   1% /run
tmpfs           3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/nvme0n1p1  8.0G  1.5G  6.6G  19% /
```

The EBS root volume contains the OS.

Your instance store volume is not yet usable. You have to format the device and mount it as follows:

```
$ sudo mkfs -t xfs /dev/nvme1n1
$ sudo mkdir /data
$ sudo mount /dev/nvme1n1 /data
```

Formats by creating an XFS filesystem

Creates a folder on which to mount the device

Mounts the device

You can now use the instance store by reading and writing to `/data` like this:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
[...]
/dev/nvme1n1    110G  145M  110G   1% /data
```

110 GB are available.

> **Windows**
>
> For Windows instances, instance store volumes are NTFS formatted and mounted automatically.

8.2.2 Testing performance

Let's take the same performance measurements we took in section 8.1.3 to see the difference between the instance store and EBS volumes, as shown next:

```
$ sudo dd if=/dev/zero of=/data/tempfile bs=1M count=1024 \
➥ conv=fdatasync,notrunc
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 13.3137 s, 80.6 MB/s
```

18% faster compared with EBS from section 8.1.3

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ sudo dd if=/data/tempfile of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 5.83715 s, 184 MB/s
sh-4.2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
```

170% faster compared with EBS from section 8.1.3

Keep in mind that performance can vary, depending on your actual workload. This example assumes a file size of 1 MB. If you're hosting websites, you'll most likely deal with lots of small files instead. The performance characteristics show that the instance store is running on the same hardware as the virtual machine. The volumes are not connected to the virtual machine over the network as with EBS volumes.

Cleaning up

Don't forget to delete your stacks after you finish this section to clean up all used resources. Otherwise, you'll be charged for the resources you use.

8.2.3 Backing up your data

There is no built-in backup mechanism for instance store volumes. Based on what you learned in section 7.3, you can use a combination of scheduled jobs and S3 to back up your data periodically, as shown here:

```
$ aws s3 sync /path/to/data s3://$YourCompany-backup/instancestore-backup
```

If you need to back up data, you should probably use more durable, block-level storage like EBS. An instance store is better used for ephemeral persistence requirements.

You will learn about another option to store your data in the next chapter: a network filesystem.

Summary

- Block-level storage can be used only in combination with an EC2 instance because the OS is needed to provide access to the block-level storage (including partitions, filesystems, and read/write system calls).
- When creating an EBS volume, you need to specify the volume size. AWS charges you for the provisioned storage, no matter whether or not you are using all the storage. Also, it is possible to increase the size of a volume later.
- EBS volumes are connected to a single EC2 instance via network. Depending on your instance type, this network connection can use more or less bandwidth.
- There are different types of EBS volumes available: General Purpose SSD (gp3), Provisioned IOPS SSD (io2), Throughput Optimized HDD (st1), and Cold HDD (sc1) are the most common.

- EBS snapshots are a powerful way to back up your EBS volumes because they use a block-level, incremental approach.
- An instance store provides low latency and high throughput. However, as you are using storage directly attached to the physical machine running your virtual machine, data is lost if you stop or terminate the instance.
- Typically, an instance store is used only for temporary data that does not need to be persisted.

9 Sharing data volumes between machines: EFS

This chapter covers

- Creating a highly available network filesystem
- Mounting a network filesystem on multiple EC2 instances
- Sharing files between EC2 instances
- Tweaking the performance of your network filesystem
- Monitoring possible bottlenecks in your network filesystem
- Backing up your shared filesystem

Many legacy applications store state in a filesystem on disk. Therefore, using Amazon S3—an object store, described in chapter 7—isn't possible without modifying the application. Using block storage as discussed in the previous chapter might be an option, but it doesn't allow you to access files from multiple machines. Because block storage persists data in a single data center only, AWS promises an uptime of only 99.9%.

If you need to share a filesystem between virtual machines or require high availability, the Elastic File System (EFS) might be an option. EFS is based on the NFSv4.1 protocol, which allows you to mount and access the filesystem on one or multiple machines in parallel. EFS distributes data among multiple data centers,

called availability zones, and promises an uptime of 99.99%. In this chapter, you learn how to set up EFS, tweak the performance, and back up your data.

> **EFS WORKS ONLY WITH LINUX** At this time, EFS isn't supported by Windows EC2 instances. The Amazon FSx for Windows File Server is an alternative to EFS for Windows workloads. See http://mng.bz/zmq1 to learn more.

Let's take a closer look at how EFS works compared to Elastic Block Store (EBS) and the instance store, introduced in chapter 8. An EBS volume is tied to a data center and can be attached to only a single EC2 instance from the same data center. Typically, EBS volumes are used as the root volumes that contain the operating system, or, for relational database systems, to store the state. An instance store consists of a hard drive directly attached to the hardware the virtual machine is running on. An instance store can be regarded as ephemeral storage and is, therefore, used only for temporary data, caching, or for systems with embedded data replication, like many NoSQL databases, for example. In contrast, the EFS filesystem supports reads and writes by multiple EC2 instances from different data centers in parallel. In addition, the data on the EFS filesystem is replicated among multiple data centers and remains available even if a whole data center suffers an outage, which isn't true for EBS and instance stores. Figure 9.1 shows the differences: EBS is a virtual disk, instance store is a local disk, and EFS is a shared folder.

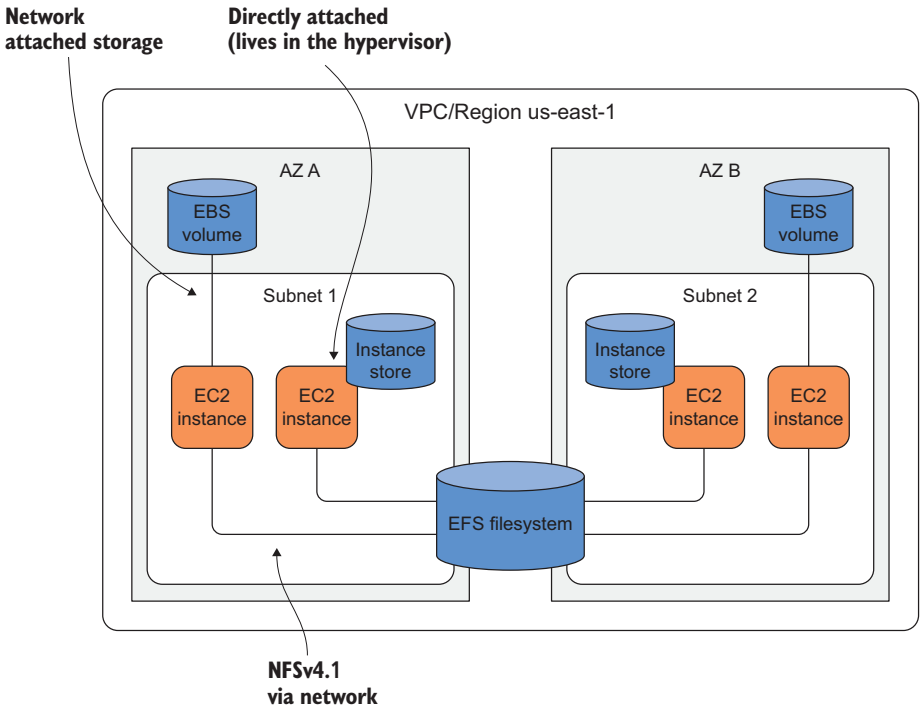


Figure 9.1 Comparing EBS, instance stores, and EFS

All examples are covered by the Free Tier

All examples in this chapter are covered by the Free Tier. As long as you follow the instructions and don't run them longer than a few days, you won't pay anything. You will find instructions to delete all resources at the end of the chapter.

EFS consists of two components:

- *Filesystem*—Stores your data
- *Mount target*—Makes your data accessible

The filesystem is the resource that stores your data in an AWS region, but you can't access it directly. To do so, you must create an EFS mount target in a subnet. The mount target provides a network endpoint that you can use to mount the filesystem on an EC2 instance via NFSv4.1. The EC2 instance must be in the same subnet as the EFS mount target, but you can create mount targets in multiple subnets. Figure 9.2 demonstrates how to access the filesystem from EC2 instances running in multiple subnets.

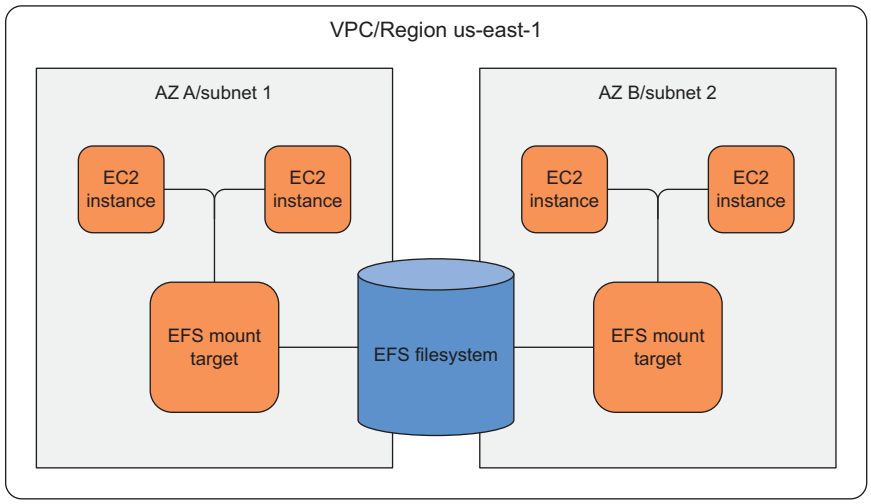


Figure 9.2 Mount targets provide an endpoint for EC2 instances to mount the filesystem in a subnet.

Equipped with the knowledge about filesystems and mount targets, we will now continue to practice.

Linux is a multiuser operating system. Many users can store data and run programs isolated from each other. Each user has a home directory, which is usually stored under /home/$username. For example, the user michael owns the home

directory /home/michael. Of course, only user michael is allowed to read and write in /home/michael. The `ls -d -l /home/*` command lists all home directories, as shown next:

```
$  ls -d -l /home/*
drwx------ 2 andreas    andreas    4096
➥ Jul 24 06:25 /home/andreas
drwx------ 3 michael    michael    4096
➥ Jul 24 06:38 /home/michael
```

Lists all home directories with absolute paths

/home/andreas is accessible by the user and group andreas only.

/home/michael can only be accessed by user and group michael.

When using multiple EC2 instances, your users will have a separate home folder on each EC2 instance. If a Linux user uploads a file on one EC2 instance, they can't access the file on another EC2 instance. To solve this problem, create a filesystem and mount EFS on each EC2 instance under /home. The home directories are then shared across all your EC2 instances, and users will feel at home no matter which machine they log in to. In the following sections, you will build this solution step-by-step. First, you will create the filesystem.

9.1 Creating a filesystem

The filesystem is the resource that stores your files, directories, and links. Like S3, EFS grows with your storage needs. You don't have to provision the storage up front. The filesystem is located in an AWS region and replicates your data under the covers across multiple data centers. You will use CloudFormation to set up the filesystem next.

9.1.1 Using CloudFormation to describe a filesystem

First, configure the filesystem. The next listing shows the CloudFormation resource.

Listing 9.1 CloudFormation snippet of an EFS filesystem resource

```
Resources:
  [...]
  FileSystem:
    Type: 'AWS::EFS::FileSystem'
    Properties:
      Encrypted: true
      ThroughputMode: bursting
      PerformanceMode: generalPurpose
      FileSystemPolicy:
        Version: '2012-10-17'
        Statement:
        - Effect: 'Deny'
          Action: '*'
```

Specifies the stack resources and their properties

We recommend to enable encryption at rest by default.

The default throughput mode is called bursting. We'll cover more on the throughput mode for I/O intensive workloads later.

The default performance mode is general purpose. We'll cover more on the performance mode later.

It's a security best practice to encrypt data in transit. The filesystem policy ensures that all access uses secure transport.

```
    Principal:
      AWS: '*'
    Condition:
      Bool:
        'aws:SecureTransport': 'false'
```

That's it. The filesystem is ready to store data. Before we do so, let's talk about the costs.

9.1.2 Pricing

Estimating costs for storing data on EFS is not that complicated. The following three factors affect the price:

- The amount of stored data in GB per month
- The frequency that the data is accessed
- Whether you want to trade in availability for cost

When accessing data frequently choose the Standard Storage or One Zone Storage storage class, which comes with the lowest latency. When accessing data less than daily, consider Standard–Infrequent Access Storage or One Zone–Infrequent Access Storage to reduce storage costs. Keep in mind that doing so increases the first-byte latency. Also, when using the Infrequent Access Storage classes, accessing the data comes with a fee.

If you do not need the high availability of 99.99% provided by the Standard Storage and Standard–Infrequent Access Storage, consider choosing the One Zone Storage or One Zone–Infrequent Access Storage storage classes. Those storage classes do not replicate your data among multiple data centers, which reduces the promised availability to 99.9%. It is worth mentioning that all storage classes are designed for a durability of 99.999999999%. However, you should back up your data stored with One Zone to be able to recover in the event of a data center destruction. You will learn more about backing up an EFS filesystem at the end of the chapter.

Table 9.1 shows EFS pricing when storing data in US East (N. Virginia), also called us-east-1.

Table 9.1 EFS storage classes affect the monthly costs for storing data.

Storage class	Price per GB/month	Access requests per GB transferred
Standard Storage	$0.30	$0.00
Standard–Infrequent Access Storage	$0.025	$0.01
One Zone Storage	$0.16	$0.00
One Zone–Infrequent Access Storage	$0.0133	$0.01

Let's briefly estimate the costs for storing 5 GB of data on EFS. Assuming the data is accessed multiple times per day and high availability is required, we choose the

Standard Storage class. So, the cost for storing data is 5 GB × $0.30, which results in a total of $1.50 a month.

Please note: the first 5 GB (Standard Storage) per month are free in the first year of your AWS account (Free Tier). For more details about EFS pricing go to https://aws.amazon.com/efs/pricing/.

After configuring an EFS filesystem with the help of CloudFormation and estimating costs, you will learn how to mount the NFS share on an EC2 instance. To do so, you need to create a mount target first.

9.2 Creating a mount target

An EFS mount target makes your data available to EC2 instances via the NFSv4.1 protocol in a subnet. The EC2 instance communicates with the mount target via a TCP/IP network connection. As you learned in section 6.4, you control network traffic on AWS using security groups. You can use a security group to allow inbound traffic to an EC2 instance or an RDS database, and the same is true for a mount target. Security groups control which traffic is allowed to enter the mount target. The NFS protocol uses port 2049 for inbound communication. Figure 9.3 shows how mount targets are protected.

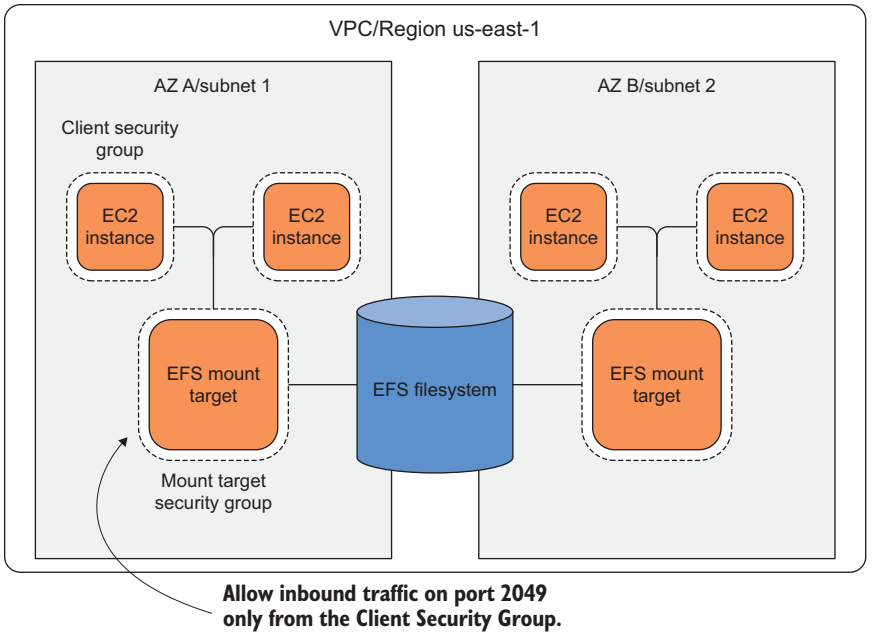


Figure 9.3 EFS mount targets are protected by security groups.

In our example, to control traffic as tightly as possible, you won't grant traffic based on IP addresses. Instead, you'll create two security groups. The client security group

will be attached to all EC2 instances that want to mount the filesystem. The mount target security group allows inbound traffic on port 2049 only for traffic that comes from the client security group. This way, you can have a dynamic fleet of clients who are allowed to send traffic to the mount targets.

EFS IS NOT ONLY ACCESSIBLE FROM EC2 INSTANCES In this chapter, we are focusing on mounting an EFS filesystem on EC2 instances. In our experience, that's the most typical use case for EFS. However, EFS can also be used in the following other scenarios:

- Containers (ECS and EKS)
- Lambda functions
- On-premises servers

Next, use CloudFormation to manage an EFS mount target. The mount target references the filesystem, needs to be linked to a subnet, and is also protected by at least one security group. You will first describe the security groups, followed by the mount target, as shown in the following listing.

Listing 9.2 CloudFormation snippet of an EFS mount target and security groups

```
Resources:
  [...]
  EFSClientSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'EFS Mount target client'
      VpcId: !Ref VPC
  MountTargetSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'EFS Mount target'
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 2049
        ToPort: 2049
        SourceSecurityGroupId: !Ref EFSClientSecurityGroup
      VpcId: !Ref VPC
  MountTargetA:
    Type: 'AWS::EFS::MountTarget'
    Properties:
      FileSystemId: !Ref FileSystem
      SecurityGroups:
      - !Ref MountTargetSecurityGroup
      SubnetId: !Ref SubnetA
```

This security group needs no rules. It's just used to mark outgoing traffic from EC2 instances.

This security group is linked to the mount target.

Allows traffic on port 2049

Only allows traffic from the security group linked to the EC2 instances

Attaches the mount target to the filesystem

Assigns the security group

Links the mount target with subnet A

Copy the `MountTargetA` resource and also create a mount target for `SubnetB` as shown in listing 9.3.

Listing 9.3 CloudFormation snippet of an EFS mount target and security groups

```
Resources:
  [...]
  MountTargetB:
    Type: 'AWS::EFS::MountTarget'
    Properties:
      FileSystemId: !Ref FileSystem
      SecurityGroups:
      - !Ref MountTargetSecurityGroup
      SubnetId: !Ref SubnetB
```

Attaches the mount target to subnet B

Next, you will finally mount the /home directory on an EC2 instance.

9.3 Mounting the EFS filesystem on EC2 instances

EFS creates a DNS name for each filesystem following the schema `$FileSystemID.efs.$Region.amazonaws.com`. From an EC2 instance, this name resolves to the mount target of the instance's subnet.

We recommend using the EFS mount helper to mount EFS filesystems because the tool comes with two features: secure transport with TLS and authentication with IAM. Besides that, the EFS mount helper applies the recommended defaults for mounting EFS filesystems.

On Amazon Linux 2, which we are using for our examples, installing the EFS mount helper is quite simple, as shown here:

```
$ sudo yum install amazon-efs-utils
```

With the EFS mount util installed, the following command mounts an EFS filesystem. This snippet shows the full mount command:

```
$ sudo mount -t efs -o tls,iam $FileSystemID $EFSMountPoint
```

Replace `$FileSystemID` with the EFS filesystem, such as `fs-123456`, and `$EFSMountPoint` with the local path where you want to mount the filesystem. The following code snippet shows an example:

```
$ sudo mount -t efs -o tls,iam fs-123456 /home
```

1. `tls` initiates a TLS tunnel from the EC2 instance to the EFS filesystem to encrypt data in transit.
2. `iam` enables authentication via IAM using the IAM role attached to the EC2 instance.

Of course, it is also possible to use the /ets/fstab config file to automatically mount on startup. Again, you need to replace `$FileSystemID` and `$EFSMountPoint` as described in the previous example:

```
$FileSystemID:/ $EFSMountPoint efs _netdev,noresvport,tls,iam 0 0
```

You are already familiar with the options `tls` and `iam`. The two other options have the following meanings:

- `_netdev`—Identifies a network filesystem
- `noresvport`—Ensures that a new TCP source port is used when reestablishing a connection, which is required when recovering from network problems

It's now time to add two EC2 instances to the CloudFormation template. Each EC2 instance should be placed in a different subnet and mount the filesystem to /home. The /home directory will exist on both EC2 instances, and it will also contain some data (such as the folder ec2-user). You have to ensure that you're copying the original data the first time before you mount the EFS filesystem, which is empty by default. The following listing describes the EC2 instance that copies the existing /home folder before the shared home folder is mounted.

Listing 9.4 Using CloudFormation to launch an EC2 instance and mount an EFS filesystem

```
Resources:
  [...]
  EC2InstanceA:                                    # Creates an EC2 instance
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      InstanceType: 't2.micro'
      IamInstanceProfile: !Ref IamInstanceProfile  # The IAM role grants access to the EFS filesystem (see the next listing).
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:                                  # Attaches the security group, which is used to identify outgoing traffic to the filesystem
        - !Ref EFSClientSecurityGroup
        SubnetId: !Ref SubnetA                     # Places the EC2 instance into subnet A
      UserData:                                    # Adds a Bash script to the user data of the virtual machine, which is executed automatically at the end of the first boot process
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}
➥ --resource EC2InstanceA --region ${AWS::Region}' ERR

          # install dependencies
          yum install -y nc amazon-efs-utils
          pip3 install botocore

          # copy existing /home to /oldhome          # Backs up all home directories to /oldhome
          mkdir /oldhome
          cp -a /home/. /oldhome

          # wait for EFS mount target                # After creating a new filesystem, it takes a few minutes until its DNS name resolves to the mount targets.
          while ! (echo > /dev/tcp/${FileSystem}.efs.${AWS::Region}.
➥ amazonaws.com/2049) >/dev/null 2>&1; do sleep 5; done

          # mount EFS filesystem
          echo "${FileSystem}:/ /home efs _netdev,noresvport,tls,iam 0 0"
```

```
➥ >> /etc/fstab
          mount -a

          # copy /oldhome to new /home
          cp -a /oldhome/. /home

          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
➥ --resource EC2InstanceA --region ${AWS::Region}
      Tags:
      - Key: Name
        Value: 'efs-a'
    CreationPolicy:
      ResourceSignal:
        Timeout: PT10M
    DependsOn:
    - VPCGatewayAttachment
    - MountTargetA
```

Adds an entry to fstab, which makes sure the filesystem is mounted automatically on each boot

Mounts all entries—most importantly the EFS filesystem—defined in fstab without the need of rebooting the system

Copies the old home directories to the EFS filesystem mounted under /home

Sends a success signal to CloudFormation

Tells CloudFormation to wait until it receives a success signal from the EC2 instance

The EC2 instance requires internet connectivity as well as a mount target. Because both dependencies are not apparent to CloudFormation, we add them manually here.

To prove that the EFS filesystem allows you to share files across multiple instances, we are adding a second EC2 instance, as shown next.

Listing 9.5 Mounting an EFS filesystem from a second EC2 instance

```
Resources:
  [...]
  EC2InstanceB:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      InstanceType: 't2.micro'
      IamInstanceProfile: !Ref IamInstanceProfile
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref EFSClientSecurityGroup
        SubnetId: !Ref SubnetB
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}
➥ --resource EC2InstanceB --region ${AWS::Region}' ERR

          # install dependencies
          yum install -y nc amazon-efs-utils
          pip3 install botocore

          # wait for EFS mount target
          while ! (echo > /dev/tcp/${FileSystem}.efs.${AWS::Region}
➥ .amazonaws.com/2049) >/dev/null 2>&1; do sleep 5; done
```

Places the EC2 instance into subnet B

The old /home folder is not copied here. This is already done on the first EC2 instance in subnet A.

```
          # mount EFS filesystem
          echo "${FileSystem}:/ /home efs _netdev,noresvport,tls,iam 0 0"
➥ >> /etc/fstab
          mount -a

          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
➥ --resource EC2InstanceB --region ${AWS::Region}
      Tags:
      - Key: Name
        Value: 'efs-b'
    CreationPolicy:
      ResourceSignal:
        Timeout: PT10M
    DependsOn:
    - VPCGatewayAttachment
    - MountTargetB
```

To make things easier, you can also add outputs to the template to expose the IDs of your EC2 instances like this:

```
Outputs:
  EC2InstanceA:
    Value: !Ref EC2InstanceA
    Description: 'Id of EC2 Instance in AZ A (connect via Session Manager)'
  EC2InstanceB:
    Value: !Ref EC2InstanceB
    Description: 'Id of EC2 Instance in AZ B (connect via Session Manager)'
```

The CloudFormation template is now complete. It contains the following:

- The EFS filesystem
- Two mount targets in subnet A and subnet B
- Security groups to control traffic from the EC2 instances to the mount targets
- EC2 instances in both subnets, including a `UserData` script to mount the filesystem

Where is the template located?

You can find the template on GitHub. Download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter09/efs.yaml. On S3, the same file is located at https://s3.amazonaws.com/awsinaction-code3/chapter09/efs.yaml.

It's now time to create a stack based on your template to create all the resources in your AWS account. Use the AWS CLI to create the stack like this:

```
$ aws cloudformation create-stack --stack-name efs \
➥ --template-url https://s3.amazonaws.com/awsinaction-code3/\
➥ chapter09/efs.yaml --capabilities CAPABILITY_IAM
```

Once the stack is in the state CREATE_COMPLETE, two EC2 instances are running. Both mounted the EFS share to /home. You also copied the existing home directories to the EFS share. It's time to connect to the instances via the Session Manager and do some tests to see whether users can really share files between the EC2 instances in their home directory.

9.4 Sharing files between EC2 instances

Use the Session Manager to connect to the virtual machine named EC2InstanceA. Use the following command to get the EC2 instance IDs of EC2InstanceA and EC2InstanceB:

```
$ aws cloudformation describe-stacks --stack-name efs \
➥ --query "Stacks[0].Outputs"
[{
  "Description": "[...]",
  "OutputKey": "EC2InstanceA",
  "OutputValue": "i-011a050b697d12e7a"
}, {
  "Description": "[...]",
  "OutputKey": "EC2InstanceB",
  "OutputValue": "i-a22b67b2a4d25a2b"
}]
```

Then, establish a second connection to the virtual machine EC2InstanceB with the help of the Session Manager and switch to the user's home directory, as shown in the next code snippet. By the way, the default user on Amazon Linux 2 is the ec2-user. But when using the Session Manager via the AWS Management Console, you are logged in with another user named ssm-user:

```
$ cd $HOME
```

Changes to the home directory of the current user

Also check whether there are any files or folders in your home directory as follows:

```
$ ls
```

If no data is returned, the folder /home/ec2-user is empty.

Now, create a file on one of the machines like this:

```
$ touch i-was-here
```

The touch command creates an empty file named i-was-here.

On the other machine, confirm that you can see the new file as follows:

```
$ cd $HOME
$ ls
i-was-here
```

The file created on the other machine appears here.

This simple experiment proves that you have access to the same home directory on both machines. You could add hundreds of machines to this example. All would share

the same home directory, and users would be able to access the same home directory on all EC2 instances. You can apply the same mechanism to share files between a fleet of web servers, for example, mounting the folder /var/www/html from an EFS filesystem. A similar example is building a highly available Jenkins server by putting /var/lib/jenkins on EFS.

To run the solution successfully, you also need to take care of backups, performance tuning, and monitoring. You will learn about this in the following sections.

9.5 Tweaking performance

EFS makes it easy for us to mount a network filesystem from many machines. However, we can say from our own experience that we repeatedly have problems with the performance of EFS in everyday practice. Therefore, in the following section, we describe what you should consider to get the maximum performance out of EFS.

The following factors affect latency, throughput, and I/O operations per second of an EFS filesystem:

- *The performance mode*—General Purpose or Max I/O
- *The throughput mode*—Bursting or Provisioned
- *The storage class*—Standard or One Zone

Let's dive into the details.

9.5.1 Performance mode

EFS comes with two performance modes:

- *General Purpose mode*—Supports up to 35,000 IOPS
- *Max I/O mode*—Supports 500,000+ IOPS

IOPS stands for read or write operations per second. When reading or writing a file, EFS accounts for one I/O operation for every 4 KB with a minimum of one I/O operation.

So, when to choose which option? So far, you've used the General Purpose performance mode, which is fine for most workloads, especially latency-sensitive ones where small files are served most of the time. The /home directory is a perfect example of such a workload. Typical files like documents are relatively small, and users expect low latency when fetching files.

But sometimes, EFS is used to store massive amounts of data for analytics. For data analytics, latency is not important. Throughput is the metric you want to optimize instead. If you want to analyze gigabytes or terabytes of data, it doesn't matter if your time to first byte takes 1 ms or 100 ms. Even a small increase in throughput will decrease the time it will take to analyze the data. For example, analyzing 1 TB of data with 100 MB/second throughput will take 167 minutes. That's almost three hours, so the first few milliseconds don't really matter. Optimizing for throughput can be achieved using the Max I/O performance mode.

Please note: the performance mode being used by an EFS filesystem cannot be changed—you set it when the filesystem is created. Therefore, to change the performance mode, you have to create a new filesystem. We recommend you start with the General Purpose performance mode if you are unsure which mode fits best for your workload. You will learn how to check whether you made the right decision by monitoring data, which is covered in the following section.

As most AWS services do, EFS sends metrics to CloudWatch, allowing us to get insight into the performance of a filesystem. We recommend creating a CloudWatch alarm to monitor filesystems with General Purpose performance mode. The metric `PercentIOLimit` tells you whether a filesystem is approaching its I/O limit. By migrating your data to a filesystem with Max I/O mode, you can increase the I/O limit from 35,000 IOPS to 500,000+ IOPS. Keep in mind, however, that doing so increases read latency from around 600 microseconds to single-digit milliseconds. The next listing shows the CloudWatch alarm to monitor the `PercentIOLimit` metric.

Listing 9.6 Monitoring the `PercentIOLimit` metric using a CloudWatch alarm

If the PercentIOLimit metric reaches 100%, we are hitting the limit of 35,000 read or 7,000 write operations. Switching to the Max I/O performance mode is a possible mitigation.

```
PercentIOLimitTooHighAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    AlarmDescription: 'I/O limit has been reached, consider ...'
    Namespace: 'AWS/EFS'
    MetricName: PercentIOLimit
    Statistic: Maximum
    Period: 600
    EvaluationPeriods: 3
    ComparisonOperator: GreaterThanThreshold
    Threshold: 95
    Dimensions:
    - Name: FileSystemId
      Value: !Ref FileSystem
```

Creates a CloudWatch alarm

EFS uses the AWS/EFS for all its metrics.

The alarm monitors a period of 600 seconds.

The alarm will trigger if the maximum PercentIOLimit was greater than the threshold.

The threshold is set to 95%.

Each filesystem reports its own PercentIOLimit metric. Therefore, we are configuring a dimension here.

References the ID of the filesystem

The alarm takes into consideration the last three periods, which is 3x 600 seconds.

The dimension is named FileSystemId.

The alarm evaluates the PercentIOLimit metric by using the maximum.

Next, you will learn about the second factor that affects the performance of an EFS filesystem: the throughput mode.

9.5.2 Throughput mode

Besides the performance mode, the throughput mode determines the maximum throughput of an EFS filesystem using the following modes:

- *Bursting Throughput mode*—Comes with a small baseline throughput but is able to burst throughput for short periods of time. Also, the throughput grows with the amount of data stored.
- *Provisioned Throughput mode*—Gives you a constant throughput with a maximum of 1 GiBps. You pay $6.00 per MB/s per month.

BURSTING THROUGHPUT MODE

When using Bursting Throughput mode, the amount of data stored in a filesystem affects the baseline and bursting throughput. The baseline throughput for an EFS filesystem is 50 MiBps per TiB of storage with a minimum of 1 MiBps. For a limited period of time, reading or writing data may burst up to 100 MiBps per TiB of storage, with a minimum of 100 MiBps.

> **WARNING** The discussed limits are labeled default limits by AWS. Depending on your scenario, it might be possible to increase those limits. However, there is no guarantee for that, and AWS does not provide any information on the maximum of a possible limit increase.

A filesystem with Bursting Throughput mode handles up to 1 or 3 GiBps, depending on the region (see http://mng.bz/09Av). Table 9.2 shows the baseline and bursting write throughput of an EFS filesystem in US East (N. Virginia) where the maximum bursting throughput is 3 GiBps.

Table 9.2 EFS throughput depends on the storage size.

Filesystem size	Baseline throughput	Bursting throughput	Explanation
<=20 GiB	1 MiBps	100 MiBps	Minimum baseline throughput is 1 MiBps, and minimum bursting throughput is 100 MiBps.
1 TiB	50 MiBps	100 MiBps	The bursting throughput starts to grow with filesystems larger than 1 TiB.
30 TiB	1500 MiBps	3000 MiBps	The bursting throughput hits the maximum for filesystems in US East (N. Virginia).
>=60 TiB	3000 MiBps	3000 MiBps	Both the baseline and bursting throughput hit the maximum for filesystems in US East (N. Virginia).

As long as you are consuming less than the baseline throughput, the filesystem accumulates burst credits. For every TiB of storage, your filesystem accrues credits of 50 MiB per second. For example, a filesystem with 500 GiB accumulates burst credits for accessing 2160 GiB within 24 hours. That's bursting at the maximum of 100 MiBps for 6 hours.

WARNING EFS does not support more than 500 MiBps per client, which means an EC2 instance cannot transfer more than 500 MiBps to an EFS filesystem.

Be aware that the maximum credit balance is 2.1 TiB for filesystems smaller than 1 TiB and 2.1 TiB per TiB stored for filesystems larger than 1 TiB. This means filesystems can burst for no more than 12 hours.

It is important to mention that a discount exists for reading data compared to writing data. A filesystem allows 1.66 or three times more read throughput depending on the region. For US East (N. Virginia), the discount is 1.66, which means the maximum read throughput is 5 GiBps. However, be aware that the discount for reading data is not valid for requests returning less than 4 KB of data.

To make things even more complicated, it is worth mentioning that data stored in the Infrequent Access storage class is not taken into consideration when calculating the maximum baseline and burst throughput.

When using the Bursting Throughput mode, you should monitor the burst credits of the filesystem because if the filesystem runs out of burst credits, the throughput will decrease significantly. We've seen many outages caused by an unexpected drop in the baseline performance. The following listing shows how to create a CloudWatch alarm to monitor the `BurstCreditBalance` metric. We've configured the threshold of the alarm to a burst credit of 192 GB, which will allow the filesystem to burst at 100 MiBps for an hour.

Listing 9.7 Monitoring the `PercentIOLimit` metric using a CloudWatch alarm

```
BurstCreditBalanceTooLowAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    AlarmDescription: 'Average burst credit balance over last ...'
    Namespace: 'AWS/EFS'
    MetricName: BurstCreditBalance
    Statistic: Average
    Period: 600
    EvaluationPeriods: 1
    ComparisonOperator: LessThanThreshold
    Threshold: 192000000000
    Dimensions:
    - Name: FileSystemId
      Value: !Ref FileSystem
```

The alarm monitors the BurstCreditBalance metric.

The threshold of 192000000000 bytes translates into 192 GB (lasts for ~30 minutes; you can burst at 100 MiBps).

If you are running out of burst credits because your workload requires a high and constant throughput, you could switch to the Provisioned Throughput mode or add data to your EFS filesystem to increase the baseline and bursting throughput.

PROVISIONED THROUGHPUT MODE

It is possible to upgrade an EFS filesystem from Bursting Throughput mode to Provisioned Throghput mode at any time. When activating the Provisioned Throughput mode, you need to specify the provisioned throughput of the filesystem in MiBps.

A filesystem will deliver the provisioned throughput continuously. The maximum provisioned throughput is 1 GiBps. Also, you have to pay for the provisioned throughput. AWS charges $6.00 per MB/s per month.

For a filesystem in Provisioned Throughput mode, you pay for throughput only above the baseline performance. So, for example, when provisioning 200 MiBps for a filesystem with 1 TiB, you will pay for only 150 MiBps, because 50 MiBps would be the baseline performance of a filesystem in Bursting Throughput mode.

The following code snippet shows how to provision throughput using CloudFormation. Use the efs-provisioned.yaml template to update your `efs` CloudFormation stack:

```
$ aws cloudformation update-stack --stack-name efs \
➥ --template-url https://s3.amazonaws.com/awsinaction-code3/\
➥ chapter09/efs-provisioned.yaml --capabilities CAPABILITY_IAM
```

The next listing contains two differences to the template you used in the previous example: the `ThroughputMode` is set to `provisioned` instead of `bursting`, and the `ProvisionedThroughputInMibps` sets the provisioned throughput to 1 MiBps, which is free, because this is the baseline throughput of a filesystem with bursting throughput.

Listing 9.8 EFS filesystem with provisioned throughput

```
FileSystem:
  Type: 'AWS::EFS::FileSystem'
  Properties:
    Encrypted: true
    ThroughputMode: provisioned
    ProvisionedThroughputInMibps: 1
    PerformanceMode: generalPurpose
    FileSystemPolicy:
      Version: '2012-10-17'
      Statement:
      - Effect: 'Deny'
        Action: '*'
        Principal:
          AWS: '*'
        Condition:
          Bool:
            'aws:SecureTransport': 'false'
```

Sets the throughput mode from bursting to provisioned

Configures the provisioned throughput in MiBps. 1 MiBps is free, even for empty filesystems.

With Provisioned Throughput mode enabled, you should keep an eye on the utilization of the throughput that you provisioned. This allows you to increase the provisioned throughput to avoid performance problems caused by EFS becoming the bottleneck of your system. Listing 9.9 shows how to create a CloudWatch alarm to monitor the utilization of the maximum throughput of an EFS filesystem. The same CloudWatch alarm works for filesystems with Bursting Throughput mode as well. This combines the metrics `MeteredIOBytes`, the throughput to the filesystem, and

`PermittedThroughput`, the maximum throughput of the filesystem, by using metric math (see http://mng.bz/qdvA for details).

Listing 9.9 Monitoring using a CloudWatch alarm and metric math

```
PermittedThroughputAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    AlarmDescription: 'Reached 80% of the permitted throughput ...'
    Metrics:
    - Id: m1
      Label: MeteredIOBytes
      MetricStat:
        Metric:
          Namespace: 'AWS/EFS'
          MetricName: MeteredIOBytes
          Dimensions:
          - Name: FileSystemId
            Value: !Ref FileSystem
        Period: 60
        Stat: Sum
        Unit: Bytes
      ReturnData: false
    - Id: m2
      Label: PermittedThroughput
      MetricStat:
        Metric:
          Namespace: 'AWS/EFS'
          MetricName: PermittedThroughput
          Dimensions:
          - Name: FileSystemId
            Value: !Ref FileSystem
        Period: 60
        Stat: Sum
        Unit: 'Bytes/Second'
      ReturnData: false
    - Expression: '(m1/1048576)/PERIOD(m1)'
      Id: e1
      Label: e1
      ReturnData: false
    - Expression: 'm2/1048576'
      Id: e2
      Label: e2
      ReturnData: false
    - Expression: '((e1)*100)/(e2)'
      Id: e3
      Label: 'Throughput utilization (%)'
      ReturnData: true
    EvaluationPeriods: 10
    DatapointsToAlarm: 6
    ComparisonOperator: GreaterThanThreshold
    Threshold: 80
```

Uses metric math to combine multiple metrics

Assigns the ID m1 to the first metric

The first metric taken into consideration is the MeteredIOBytes metric, which contains the current utilization.

The second metric is the PermittedThroughput metric, which indicates the maximum throughput.

The first metric, m1, returns the sum of bytes transferred within 60 seconds. This expression converts this into MiBps.

Assigns the ID e1 to the metric to reference it in the following line

Converts the second metric from bytes/second into MiBps

Calculates the throughput utilization in percent

The output of the third expression is the only metric/expression returning data used for the alarm.

The alarm triggers if the throughput utilization is greater than …

…80%.

If the CloudWatch alarm flips into the `ALARM` state, you might want to increase the provisioned throughput of your filesystem.

9.5.3 Storage class affects performance

The third factor that affects the performance of an EFS filesystem is the storage class used to persist data. By default, read latency to data persisted with standard storage class is as low as 600 microseconds. The latency for write requests is in the low single-digit milliseconds.

But if you choose the One Zone storage class to reduce costs, the read and write latency increases to double-digit milliseconds. Depending on your workload, this might have a significant effect on performance.

9.6 Backing up your data

AWS promises a durability of 99.999999999% (11 9s) over a given year for data stored on EFS. When using the Standard storage class, EFS replicates data among multiple data centers. Still, we highly recommend backing up the data stored on EFS, particularly so you can recover from a scenario where someone, maybe even you, accidentally deleted data from EFS.

Luckily, AWS provides a service to back up and restore data: AWS Backup. This service to centralize data protection supports EC2, EBS, S3, RDS, DynamoDB, EFS, and many more. The following three components are needed to create snapshots of an EFS filesystem with AWS Backup, as illustrated in figure 9.4:

- *Vault*—A container for grouping backups.
- *Plan*—Defines when and how to backup resources.
- *Recovery Point*—Contains all the data needed to restore a resource; you could also call it a snapshot.

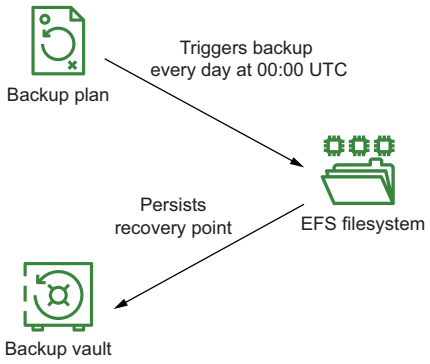


Figure 9.4 AWS Backup creates recovery points based on a schedule and stores them in a vault.

To enable backups for your EFS, update your CloudFormation stack with the template efs-backup.yaml, which contains the configuration for AWS Backups, as shown here. We'll dive into that next:

```
$ aws cloudformation update-stack --stack-name efs \
➥ --template-url https://s3.amazonaws.com/awsinaction-code3/\
➥ chapter09/efs-backup.yaml --capabilities CAPABILITY_IAM
```

Where is the template located?

You can find the template on GitHub. Download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter09/efs-backup.yaml. On S3, the same file is located at https://s3.amazonaws.com/awsinaction-code3/chapter09/efs-backup.yaml.

The following listing is an excerpt from efs-backup.yaml and shows how to create a backup vault and plan. On top of that, it includes a backup selection, which references the EFS filesystem.

Listing 9.10 Backing up an EFS filesystem with AWS Backup

```
BackupVault:
  Type: 'AWS::Backup::BackupVault'
  Properties:
    BackupVaultName: !Ref 'AWS::StackName'
BackupPlan:
  Type: 'AWS::Backup::BackupPlan'
  Properties:
    BackupPlan:
      BackupPlanName: 'efs-daily'
      BackupPlanRule:
      - RuleName: 'efs-daily'
        TargetBackupVault: !Ref BackupVault
        Lifecycle:
          DeleteAfterDays: 30
        ScheduleExpression: 'cron(0 5 ? * * *)'
        CompletionWindowMinutes: 1440
BackupSelection:
  Type: 'AWS::Backup::BackupSelection'
  Properties:
    BackupPlanId: !Ref BackupPlan
    BackupSelection:
      SelectionName: 'efs'
      IamRoleArn: !GetAtt 'BackupRole.Arn'
      Resources:
      - !GetAtt  'FileSystem.Arn'
```

Creates a backup vault to persist recovery points

A backup vault requires a unique name; therefore, we are using the name of the CloudFormation stack here.

Creates a backup plan

Each backup plan includes at least one rule.

The life cycle is configured in a way to delete backups older than 30 days.

The rule defines which backup vault should be used to store backups.

The backup plan schedules a backup for 05:00 UTC every day.

The backup should complete within 24 hours; otherwise, it will be cancelled.

Selects the resources to include in the backup

References the backup plan

An IAM role is required to grant access to EFS (see the following listing).

Adds the filesystem to the backup selection by using its ARN

Besides this, an IAM role granting access to the EFS filesystem is needed, as shown in the next listing.

Listing 9.11 The IAM role granting AWS Backup access to EFS

```
BackupRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Principal:
          Service: 'backup.amazonaws.com'
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: backup
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Action:
          - 'elasticfilesystem:Backup'
          - 'elasticfilesystem:DescribeTags'
          Resource: !Sub 'arn:${AWS::Partition}:elasticfilesystem
➥ :${AWS::Region}:${AWS::AccountId}:
➥ file-system/${FileSystem}'
```

Only the AWS Backup service is allowed to assume this IAM role.

The role grants access to create backups of EFS filesystems…

…and allows you to describe tags of EFS filesystems, which is needed in case you define a backup selection based on tags.

The policy restricts access to the filesystem we defined in the CloudFormation template earlier.

AWS Backup allows you to restore an entire filesystem or only some of the folders. It is possible to restore data to the original filesystem as well as to create a new filesystem to do so.

Unfortunately, it will take a while until AWS Backup creates a recovery point for your EFS filesystem. If you are interested in the results, wait 24 hours and check the results as follows:

1. Open the AWS Management Console.
2. Go to the AWS Backup service.
3. Open the backup vault named `efs`.
4. You'll find a list with available recovery points.
5. From here, you could restore an EFS filesystem in case of an emergency.

That's all you need to back up data stored on an EFS filesystem. We should briefly talk about the costs. You are paying for the data stored and if you need to restore your data. The following prices are valid for US East (N. Virginia). For further details, visit https://aws.amazon.com/backup/pricing/:

- Storage: $0.05 per GB and month
- Restore: $0.02 per GB

Cleaning up

First, make sure that the backup vault `efs` does not contain any recovery points like this:

```
$ aws backup list-recovery-points-by-backup-vault --backup-vault-name efs \
➥ --query "RecoveryPoints[].RecoveryPointArn"
```

If you are getting a list of recovery points, delete each of them using the following command. Don't forget to replace `$RecoveryPointArn` with the ARN of the recovery point you are trying to delete:

```
$ aws backup delete-recovery-point --backup-vault-name efs \
➥ --recovery-point-arn $RecoveryPointArn
```

After that, use the following command to delete the CloudFormation stack named `efs` that you used while going through the examples within this chapter:

```
$ aws cloudformation delete-stack --stack-name efs
```

Summary

- EFS provides a NFSv4.1-compliant filesystem that can be shared between Linux EC2 instances in different availability zones.
- EFS mount targets are bound to an availability zone and are protected by security groups.
- Data that is stored in EFS is replicated across multiple data centers.
- EFS comes with two performance modes: General Purpose and Max I/O. Use General Purpose if your workload accesses many small files, and Max I/O if you have a few large files.
- By default, the throughput of an EFS filesystem is capable of bursting for a certain amount of time. If you are expecting a high and constant throughput, you should use provisioned throughput instead.
- You need at least two mount targets in different data centers (also called availability zones) for high availability.
- Encryption of data in transit (TLS) and IAM authentication help to protect sensitive data stored on EFS.
- AWS Backup allows you to create and restore snapshots of EFS filesystems.

10 Using a relational database service: RDS

This chapter covers

- Launching and initializing relational databases with RDS
- Creating and restoring database snapshots
- Setting up a highly available database
- Monitoring database metrics
- Tweaking database performance

WordPress is a content management system that powers substantial parts of the internet. Like many other applications, WordPress uses a relational database to store articles, comments, users, and many other data. It is fair to say that relational databases are the de facto standard for storing and querying structured data, and many applications are built on top of a relational database system such as MySQL. Typically, relational databases focus on data consistency and guarantee ACID (atomicity, consistency, isolation, and durability) database transactions. A typical task is storing and querying structured data, such as the accounts and transactions in an accounting application. If you want to use a relational database on AWS, you have two options:

- Use the managed relational database service *Amazon RDS*, which is offered by AWS.
- Operate a relational database yourself on top of virtual machines.

The Amazon Relational Database Service (Amazon RDS) offers ready-to-use relational databases such as PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server. Beyond that, AWS offers its own engine called Amazon Aurora, which is MySQL and PostgreSQL compatible. As long as your application uses one of these database systems, it is not a big deal to switch to RDS. The trickiest part is migrating the data, which you will learn about in this chapter as well.

RDS is a managed service. The managed service provider—in this case, AWS—is responsible for providing a defined set of services—in this case, operating a relational database system. Table 10.1 compares using an RDS database and hosting a database yourself on virtual machines.

Table 10.1 Managed service RDS vs. a self-hosted database on virtual machines

	Amazon RDS	Self-hosted on virtual machines
Cost for AWS services	Higher because RDS costs more than virtual machines (EC2)	Lower because virtual machines (EC2) are cheaper than RDS
Total cost of ownership	Lower because operating costs are split among many customers	Much higher because you need your own manpower to manage your database
Quality	AWS professionals are responsible for the managed service.	You'll need to build a team of professionals and implement quality control yourself.
Flexibility	High, because you can choose a relational database system and most of the configuration parameters	Higher, because you can control every part of the relational database system you installed on virtual machines

You'd need considerable time and know-how to build a comparable relational database environment based on virtual machines, so we recommend using Amazon RDS for relational databases whenever possible to decrease operational costs and improve quality. That's why we won't cover hosting your own relational database on VMs in this book. Instead, we'll introduce Amazon RDS in detail.

In this chapter, you'll launch a MySQL database with the help of Amazon RDS. Chapter 2 introduced a WordPress setup like the one shown in figure 10.1 and described next; you'll use this example in this chapter, focusing on the database part:

1. The user sends an HTTP request to the load balancer.
2. The load balancer distributes the incoming request to a fleet of virtual machines.
3. Each virtual machine runs a web server, which connects to a MySQL database as well as a network filesystem.

After the MySQL database is up and running, you'll learn how to import, back up, and restore data. More advanced topics like setting up a highly available database and improving the performance of the database will follow.

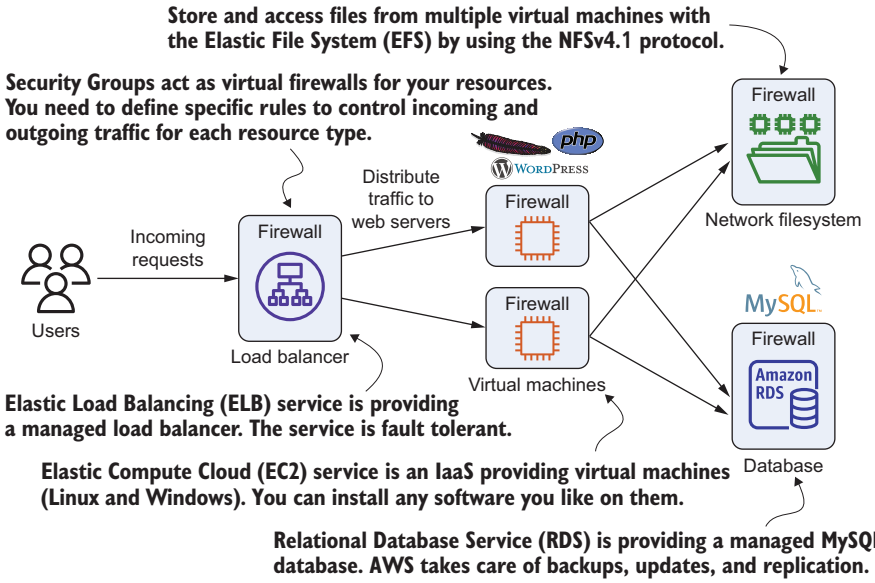


Figure 10.1 The company's blogging infrastructure consists of two load-balanced web servers running WordPress and a MySQL database server.

> **Not all examples are covered by the Free Tier**
>
> The examples in this chapter are not all covered by the Free Tier. A warning message appears when an example incurs costs. Nevertheless, as long as you don't run all other examples longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

10.1 Starting a MySQL database

In the following section, you will launch the infrastructure required to run WordPress on AWS. In this chapter, we will focus on the MySQL database provided by RDS, but you can easily transfer what you learn to other database engines such as Aurora, PostgreSQL, MariaDB, Oracle, and Microsoft SQL Server, as well as to applications other than WordPress.

When you follow the official "How to Install WordPress" tutorial (see http://mng.bz/G1vV), one of the first steps is setting up a MySQL database. Formerly, you might have installed the database system on the same virtual machine that also runs the web server. However, operating a database system is not trivial. You have to implement a solid backup and recovery strategy, for example. Also, when running a database on a single virtual machine, you are introducing a single point of failure into your system that could cause downtimes of your website.

To overcome these challenges, you will use a fully managed MySQL database provided by RDS. AWS provides backup and restore functionality and offers database systems distributed among two data centers as well as the ability to recover from failure automatically.

10.1.1 Launching a WordPress platform with an RDS database

Launching a database consists of two steps:

1. Launching a database instance
2. Connecting an application to the database endpoint

Next, you'll use the same CloudFormation template you used in chapter 2 to spin up the cloud infrastructure for WordPress. The template can be found on GitHub and on S3. You can download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter10/template.yaml. On S3, the same file is located at http://s3.amazonaws.com/awsinaction-code3/chapter10/template.yaml.

Execute the following command to create a CloudFormation stack containing an RDS database instance with a MySQL engine and web servers serving the WordPress application:

```
$ aws cloudformation create-stack --stack-name wordpress --template-url \
➥ https://s3.amazonaws.com/awsinaction-code3/chapter10/template.yaml \
➥ --parameters "ParameterKey=WordpressAdminPassword,ParameterValue=test1234" \
➥ --capabilities CAPABILITY_IAM
```

You'll have to wait several minutes while the CloudFormation stack is created in the background, which means you'll have enough time to learn the details of the RDS database instance while the template is launching. The next listing shows parts of the CloudFormation template used to create the `wordpress` stack. Table 10.2 shows the attributes you need when creating an RDS database instance using CloudFormation or the Management Console.

Table 10.2 Attributes needed to connect to an RDS database

Attribute	Description
`AllocatedStorage`	Storage size of your database in GB
`DBInstanceClass`	Size (also known as instance type) of the underlying virtual machine
`Engine`	Database engine (Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, or Microsoft SQL Server) you want to use
`DBName`	Identifier for the database
`MasterUsername`	Name for the admin user
`MasterUserPassword`	Password for the admin user

It is possible to configure a database instance as publicly accessible, but we generally do not recommend enabling access from the internet to your database, to prevent unwanted access. Instead, as shown in listing 10.1, an RDS instance should be accessible only within the VPC.

To connect to an RDS database instance, you need an EC2 instance running in the same VPC. First, connect to the EC2 instance. From there, you can then connect to the database instance.

Listing 10.1 Excerpt from the CloudFormation template for setting up an RDS database

```
Resources:
  # [...]
  DatabaseSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'awsinaction-db-sg'
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 3306
        ToPort: 3306
        SourceSecurityGroupId: !Ref WebServerSecurityGroup
  Database:
    Type: 'AWS::RDS::DBInstance'
    DeletionPolicy: Delete
    Properties:
      AllocatedStorage: 5
      BackupRetentionPeriod: 0
      DBInstanceClass: 'db.t2.micro'
      DBName: wordpress
      Engine: MySQL
      MasterUsername: wordpress
      MasterUserPassword: wordpress
      VPCSecurityGroups:
      - !Sub ${DatabaseSecurityGroup.GroupId}
      DBSubnetGroupName: !Ref DBSubnetGroup
    DependsOn: VPCGatewayAttachment
  DBSubnetGroup:
    Type: 'AWS::RDS::DBSubnetGroup'
    Properties:
      Description: DB subnet group
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
```

Security group for the database instance, allowing incoming traffic on the MySQL default port for web servers

The default MySQL port is 3306.

References the security group of EC2 instances running a web server

Disables backups. (Set to "Snapshot" for production workloads.)

Creates a database instance with Amazon RDS

The database provides 5 GB of storage.

Disables backups. (Turn this on in production.)

Creates a default database named wordpress

The size of the database instance is t2.micro, the smallest available size.

Uses MySQL as the database engine

The username for the admin user of the MySQL database

The password for the admin user of the MySQL database

References the security group for the database instance

The subnet group …

Defines the subnets the RDS database instance will launch into

… consists of subnet A and B, so RDS will distribute database instances among those subnets.

See if the CloudFormation stack named `wordpress` has reached the state `CREATE_COMPLETE` with the following command:

```
$ aws cloudformation describe-stacks --stack-name wordpress
```

Search for `StackStatus` in the output, and check whether the status is `CREATE_COMPLETE`. If not, you need to wait a few minutes longer (it can take up to 15 minutes to create the stack) and rerun the command. If the status is `CREATE_COMPLETE`, you'll find the key `OutputKey` in the output section. The corresponding `OutputValue` contains the URL for the WordPress blogging platform. The following listing shows the output in detail. Open this URL in your browser; you'll find a running WordPress setup.

Listing 10.2 Checking the state of the CloudFormation stack

```
$ aws cloudformation describe-stacks --stack-name wordpress
{
  "Stacks": [{
    "StackId": "[...]",
    "Description": "AWS in Action: chapter 10",
    "Parameters": [...],
    "Tags": [],
    "Outputs": [
        {
            "Description": "WordPress URL",
            "OutputKey": "URL",
            "OutputValue": "http://[...].us-east-1.elb.amazonaws.com"
        }
    ],
    "CreationTime": "2017-10-19T07:12:28.694Z",
    "StackName": "wordpress",
    "NotificationARNs": [],
    "StackStatus": "CREATE_COMPLETE",
    "DisableRollback": false
  }]
}
```

Opens this URL in your browser to open the WordPress application

Waits for state CREATE_COMPLETE for the CloudFormation stack

Launching and operating a relational database like MySQL is that simple. Of course, you can also use the Management Console (https://console.aws.amazon.com/rds/) to launch an RDS database instance instead of using a CloudFormation template. RDS is a managed service, and AWS handles most of the tasks necessary to operate your database in a secure and reliable way. You need to do only two things:

- Monitor your database's available storage, and make sure you increase the allocated storage as needed.
- Monitor your database's performance, and make sure you increase I/O and computing performance as needed.

Both tasks can be handled with the help of CloudWatch monitoring, as you'll learn later in the chapter.

10.1.2 Exploring an RDS database instance with a MySQL engine

The CloudFormation stack created an RDS database instance with the MySQL engine. Each database instance offers an endpoint for clients. Clients send their SQL queries to this endpoint to query, insert, delete, or update data. For example, to retrieve all

rows from a table, an application sends the following SQL request: `SELECT * FROM table`. You can request the endpoint and detailed information of an RDS database instance with the `describe-db-instances` command:

```
$ aws rds describe-db-instances --query "DBInstances[0].Endpoint"
{
  "HostedZoneId": "Z2R2ITUGPM61AM",
  "Port": 3306,
  "Address": "wdwcoq2o8digyr.cqrxioeaavmf.us-east-1.rds.amazonaws.com"
}
```

Port number of database endpoint

Host name of database endpoint

The RDS database is now running, but what does it cost?

10.1.3 Pricing for Amazon RDS

What does it cost to host WordPress on AWS? We discussed this question in chapter 2 in detail. Here, we want to focus on the costs for RDS.

Databases on Amazon RDS are priced according to the size of the underlying virtual machine and the amount and type of allocated storage. Compared to a database running on a plain EC2 VM, the hourly price of an RDS instance is higher. In our opinion, the Amazon RDS service is worth the extra charge because you don't need to perform typical DBA tasks like installation, patching, upgrades, migration, backups, and recovery.

Table 10.3 shows a pricing example for a medium-sized RDS database instance with a standby instance for high availability. All prices in USD are for US East (N. Virginia) as of March 11, 2022. Get the current prices at https://aws.amazon.com/rds/pricing/.

Table 10.3 Monthly costs for a medium-sized RDS instance

Description	Monthly price
Database instance `db.t4g.medium`	$94.17 USD
50 GB of general purpose (SSD)	$11.50 USD
Additional storage for database snapshots (100 GB)	$9.50 USD
Total	$115.17 USD

You've now launched an RDS database instance for use with a WordPress web application. You'll learn about importing data to the RDS database in the next section.

10.2 Importing data into a database

Imagine you are already running WordPress on a virtual machine in your on-premises data center, and you have decided to move the application to AWS. To do so, you need to move the data from the on-premises MySQL database to RDS. You will learn how to do that in this section.

A database without data isn't useful. In many cases, you'll need to import data into a new database by importing a dump from the old database. This section will guide you through the process of importing a MySQL database dump to an RDS database with a MySQL engine. The process is similar for all other database engines (Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server).

To import a database from your on-premises environment to Amazon RDS, follow these steps:

1. Export the database.
2. Start a virtual machine in the same region and VPC as the RDS database.
3. Upload the database dump to the virtual machine.
4. Run an import of the database dump to the RDS database on the virtual server.

We'll skip the first step of exporting a MySQL database, because the RDS instance we created in our example is empty and you may not have access to an existing WordPress database. The next sidebar gives you hints on how to create a database dump in case you need that for your real-world systems later.

Exporting a MySQL database

MySQL (and every other database system) offers a way to export and import databases. We recommend using the command-line tools from MySQL for exporting and importing databases. You may need to install the MySQL client, which comes with the `mysqldump` tool.

The following command exports all databases from localhost and dumps them into a file called dump.sql. Replace `$UserName` with the MySQL admin user, and enter the password when prompted:

```
$ mysqldump -u $UserName -p --all-databases > dump.sql
```

You can also specify only some databases for the export, as shown next. To do so, replace `$DatabaseName` with the name of the database you want to export:

```
$ mysqldump -u $UserName -p $DatabaseName > dump.sql
```

And, of course, you can export a database over a network connection as follows. To connect to a server to export a database, replace `$Host` with the host name or IP address of your database:

```
$ mysqldump -u $UserName -p $DatabaseName --host $Host > dump.sql
```

See the MySQL documentation if you need more information about the `mysqldump` tool.

Theoretically, you could import a database to RDS from any machine from your on-premises or local network, but the higher latency over the internet or VPN connection will slow down the import process dramatically. Because of this, we recommend

adding a second step: upload the database dump to a virtual machine running in the same AWS region and VPC, and import the database into RDS from there.

> **AWS Database Migration Service**
> When migrating a huge database to AWS with minimal downtime, the Database Migration Service (DMS) can help. We do not cover DMS in this book, but you can learn more on the AWS website: https://aws.amazon.com/dms.

To do so, we'll guide you through the following steps:

1. Connect to the virtual machine that is running WordPress.
2. Download a database dump from S3 to the VM. (If you are using your own database dump, we recommend uploading it to S3 first.)
3. Import the database dump into the RDS database from the virtual machine.

Fortunately, you already started two virtual machines that you know can connect to the MySQL database on RDS, because they're running the WordPress application. Go through the following steps to open a terminal session:

1. Open the EC2 service via AWS Management Console: https://console.aws.amazon.com/ec2/.
2. Select one of the two EC2 instances named wordpress.
3. Click the Connect button.
4. Select Session Manager and click the Connect button.

Because you are connected to a virtual machine with access to the RDS database instance, you are ready to import the database dump. First, change into the home directory of the `ssm-user` as follows:

```
$ cd /home/ssm-user/
```

We prepared a MySQL database dump of a WordPress blog as an example. The dump contains a blog post and a few comments. Download this database dump from S3 using the following command on the virtual machine:

```
$ wget https://s3.amazonaws.com/awsinaction-code3/chapter10/wordpress-import.sql
```

Next, you'll need the port and hostname, also called the *endpoint*, of the MySQL database on RDS. Don't remember the endpoint? The following command will print it out for you. Run this on your local machine:

```
$ aws rds describe-db-instances --query "DBInstances[0].Endpoint"
```

Run the following command on the VM to import the data from the file wordpress-import.sql into the RDS database instance; replace `$DBAddress` with the `Address` you printed to the terminal with the previous command. The `Address` will look similar to

`wdtq7tf5caejft.cd0o57zo3ohr.us-east-1.rds.amazonaws.com`. Also, type in `wordpress` when asked for a password:

```
$ mysql --host $DBAddress --user wordpress -p < wordpress-import.sql
```

Point your browser to the WordPress blog again, and you'll now find many new posts and comments there. If you don't remember the URL, run the following command on your local machine to fetch it again:

```
$ aws cloudformation describe-stacks --stack-name wordpress \
➥ --query "Stacks[0].Outputs[0].OutputValue" --output text
```

10.3 Backing up and restoring your database

Over the years, your WordPress site has accumulated hundreds of blog posts and comments from the community. That's a valuable asset. Therefore, it is key that you back up the data.

Amazon RDS is a managed service, but you still need backups of your database in case something or someone harms your data and you need to restore it, or you need to duplicate a database in the same or another region. RDS offers manual and automated *snapshots* for recovering RDS database instances. In this section, you'll learn how to use RDS snapshots to do the following:

- Configuring the retention period and time frame for automated snapshots
- Creating snapshots manually
- Restoring snapshots by starting new database instances based on a snapshot
- Copying a snapshot to another region for disaster recovery or relocation

10.3.1 Configuring automated snapshots

The RDS database you started in section 10.1 can automatically create snapshots if the `BackupRetentionPeriod` is set to a value between `1` and `35`. This value indicates how many days the snapshot will be retained (the default is `1`). Automated snapshots are created once a day during the specified time frame. If no time frame is specified, RDS picks a random 30-minute time frame during the night. A new random time frame will be chosen each night.

Creating a snapshot requires all disk activity to be briefly frozen. Requests to the database may be delayed or even fail because of a timeout, so we recommend that you choose a time frame for the snapshot that has the least effect on applications and users (e.g., late at night). Automated snapshots are your backup in case something unexpected happens to your database. This could be a query that deletes all your data accidentally or a hardware failure that causes data loss.

The following command changes the time frame for automated backups to 05:00–06:00 UTC and the retention period to three days. Use the terminal on your local machine to execute it:

```
$ aws cloudformation update-stack --stack-name wordpress --template-url \
➥ https://s3.amazonaws.com/awsinaction-code3/chapter10/\
➥ template-snapshot.yaml \
➥ --parameters ParameterKey=WordpressAdminPassword,UsePreviousValue=true \
➥ --capabilities CAPABILITY_IAM
```

The RDS database will be modified based on a slightly modified CloudFormation template, as shown in the following listing.

Listing 10.3 Modifying an RDS database's snapshot time frame and retention time

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.t2.micro'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
  DependsOn: VPCGatewayAttachment
```

Keeps snapshots for three days

Creates snapshots automatically between 05:00 and 06:00 UTC

If you want to disable automated snapshots, you need to set the retention period to `0`. As usual, you can configure automated backups using CloudFormation templates, the Management Console, or SDKs. Keep in mind that automated snapshots are deleted when the RDS database instance is deleted. Manual snapshots stay. You'll learn about them next.

10.3.2 Creating snapshots manually

You can trigger manual snapshots whenever you need, for example, before you update to the latest WordPress version, migrate a schema, or perform some other activity that could damage your database. To create a snapshot, you have to know the instance identifier. The following command extracts the instance identifier from the first RDS database instance:

```
$ aws rds describe-db-instances --output text \
➥ --query "DBInstances[0].DBInstanceIdentifier"
```

The next command creates a manual snapshot called `wordpress-manual-snapshot`. Replace `$DBInstanceIdentifier` with the output of the previous command:

```
$ aws rds create-db-snapshot --db-snapshot-identifier \
➥ wordpress-manual-snapshot \
➥ --db-instance-identifier $DBInstanceIdentifier
```

In case you get a "Cannot create a snapshot because the database instance .. is not currently in the available state." error, retry after five minutes—your database is still initializing.

It will take a few minutes for the snapshot to be created. You can check the current state of the snapshot with this command:

```
$ aws rds describe-db-snapshots \
➥ --db-snapshot-identifier wordpress-manual-snapshot
```

RDS doesn't delete manual snapshots automatically; you need to delete them yourself if you don't need them any longer. You'll learn how to do this at the end of the section.

Copying an automated snapshot as a manual snapshot

There is a difference between automated and manual snapshots. Automated snapshots are deleted automatically after the retention period is over, but manual snapshots aren't. If you want to keep an automated snapshot even after the retention period is over, you have to copy the automated snapshot to a new manual snapshot.

Get the snapshot identifier of an automated snapshot from the RDS database you started in section 10.1 by running the following command at your local terminal. Replace `$DBInstanceIdentifier` with the output of the `describe-db-instances` command:

```
$ aws rds describe-db-snapshots --snapshot-type automated \
➥ --db-instance-identifier $DBInstanceIdentifier \
➥ --query "DBSnapshots[0].DBSnapshotIdentifier" \
➥ --output text
```

The next command copies an automated snapshot to a manual snapshot named `wordpress-copy-snapshot`. Replace `$SnapshotId` with the output from the previous command:

```
$ aws rds copy-db-snapshot \
➥ --source-db-snapshot-identifier $SnapshotId  \
➥ --target-db-snapshot-identifier wordpress-copy-snapshot
```

The copy of the automated snapshot is named `wordpress-copy-snapshot`. It won't be removed automatically.

10.3.3 Restoring a database

Imagine a scary scenario: you have accidentally deleted all of the blog posts from your WordPress site. Of course, you want to restore the data as fast as possible. Lucky for you, RDS has you covered.

If you restore a database from an automated or manual snapshot, a new database will be created based on the snapshot. As figure 10.2 shows, you can't restore a snapshot to an existing database.

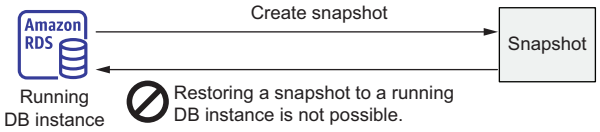


Figure 10.2 A snapshot can't be restored into an existing database.

Instead, a new database is created when you restore a database snapshot, as figure 10.3 illustrates.

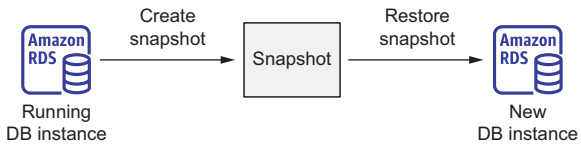


Figure 10.3 A new database is created to restore a snapshot.

To create a new database in the same VPC as the WordPress platform you started in section 10.1, you need to find out the existing database's subnet group. Execute this command to do so:

```
$ aws cloudformation describe-stack-resource \
➥ --stack-name wordpress --logical-resource-id DBSubnetGroup \
➥ --query "StackResourceDetail.PhysicalResourceId" --output text
```

You're now ready to create a new database based on the manual snapshot you created at the beginning of this section. Execute the following command, replacing `$Subnet-Group` with the output of the previous command:

```
$ aws rds restore-db-instance-from-db-snapshot \
➥ --db-instance-identifier awsinaction-db-restore \
➥ --db-snapshot-identifier wordpress-manual-snapshot \
➥ --db-subnet-group-name $SubnetGroup
```

You might get an "DBSnapshot must have state available but actually has creating" error if your snapshot has not been created yet. In this case, retry the command after five minutes.

A new database named awsinaction-db-restore is created based on the manual snapshot. In theory, after the database is created, you could switch the WordPress application to the new endpoint by modifying the /var/www/html/wp-config.php file on both virtual machines.

If you're using automated snapshots, you can also restore your database from a specified moment, because RDS keeps the database's change logs. This allows you to jump back to any point in time from the backup retention period to the last five minutes.

Execute the following command, replacing `$DBInstanceIdentifier` with the output of the earlier `describe-db-instances` command, `$SubnetGroup` with the output

of the earlier `describe-stack-resource` command, and `$Time` with a UTC timestamp from five minutes ago (e.g., `2022-03-11T09:30:00Z`):

```
$ aws rds restore-db-instance-to-point-in-time \
➥ --target-db-instance-identifier awsinaction-db-restore-time \
➥ --source-db-instance-identifier $DBInstanceIdentifier \
➥ --restore-time $Time --db-subnet-group-name $SubnetGroup
```

A new database named awsinaction-db-restore-time is created based on the source database from five minutes ago. In theory, after the database is created, you could switch the WordPress application to the new endpoint by modifying the /var/www/html/wp-config.php file on both virtual machines.

10.3.4 Copying a database to another region

When you created the cloud infrastructure for WordPress, you assumed that most readers will come from the United States. It turns out, however, that most readers access your site from Europe. Therefore, you decide to move your cloud infrastructure to reduce latency for the majority of your readers.

Copying a database to another region is possible with the help of snapshots as well. The main reasons you might do so follow:

- *Disaster recovery*—You can recover from an unlikely region-wide outage.
- *Relocating*—You can move your infrastructure to another region so you can serve your customers with lower latency.

The second command copies the snapshot named `wordpress-manual-snapshot` from the region `us-east-1` to the region `eu-west-1`. You need to replace `$SourceSnapshotArn` with the Amazon Resource Name (ARN) of the snapshot. Use the following command to get the ARN of your manual snapshot:

```
$ aws rds describe-db-snapshots \
➥ --db-snapshot-identifier wordpress-manual-snapshot \
➥ --query "DBSnapshots[0].DBSnapshotArn" --output text
```

> **COMPLIANCE** Moving data from one region to another may violate privacy laws or compliance rules. Make sure you're allowed to copy the data to another region if you're working with real data.

```
$ aws rds copy-db-snapshot \
➥ --source-db-snapshot-identifier $SourceSnapshotArn \
➥ --target-db-snapshot-identifier wordpress-manual-snapshot \
➥ --region eu-west-1
```

After the snapshot has been copied to the region `eu-west-1`, you can restore a database from it as described in the previous section.

10.3.5 Calculating the cost of snapshots

Snapshots are billed based on the storage they use. You can store snapshots up to the size of your database instance for free. In our WordPress example, you can store up to 5 GB of snapshots for free. On top of that, you pay per GB per month of used storage. As we're writing this book, the cost is $0.095 for each GB every month.

Cleaning up

It's time to clean up the snapshots and delete the restored database instances. Execute the following commands step by step, or jump to the shortcuts for Linux and macOS after the listing:

```
$ aws rds delete-db-instance --db-instance-identifier \
➥ awsinaction-db-restore --skip-final-snapshot
$ aws rds delete-db-instance --db-instance-identifier \
➥ awsinaction-db-restore-time --skip-final-snapshot
$ aws rds delete-db-snapshot --db-snapshot-identifier \
➥ wordpress-manual-snapshot
$ aws rds delete-db-snapshot --db-snapshot-identifier \
➥ wordpress-copy-snapshot
$ aws --region eu-west-1 rds delete-db-snapshot --db-snapshot-identifier \
➥ wordpress-manual-snapshot
```

Deletes the database with data from the snapshot restore

Deletes the database with data from the point-in-time restore

Deletes the manual snapshot

Deletes the copied snapshot

Deletes the snapshot copied to another region

You can avoid typing these commands manually at your terminal by using the following command to download a bash script and execute it directly on your local machine. The bash script contains the same steps as shown in the previous snippet:

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
➥ code3/main/chapter10/cleanup.sh | bash -ex
```

Keep the rest of the setup, because you'll use it in the following sections.

10.4 Controlling access to a database

Every day, you can read about WordPress sites that got hacked. One essential aspect of protecting your WordPress site is controlling access to your cloud infrastructure and database.

The shared-responsibility model applies to the RDS service as well as to AWS services in general. AWS is responsible for the security of the cloud in this case—for example, for the security of the underlying OS. You, the customer, need to specify the rules controlling access to your data and RDS database.

Figure 10.4 shows the following three layers that control access to an RDS database:

- Controlling access to the configuration of the RDS database
- Controlling network access to the RDS database
- Controlling data access with the database's own user and access management features

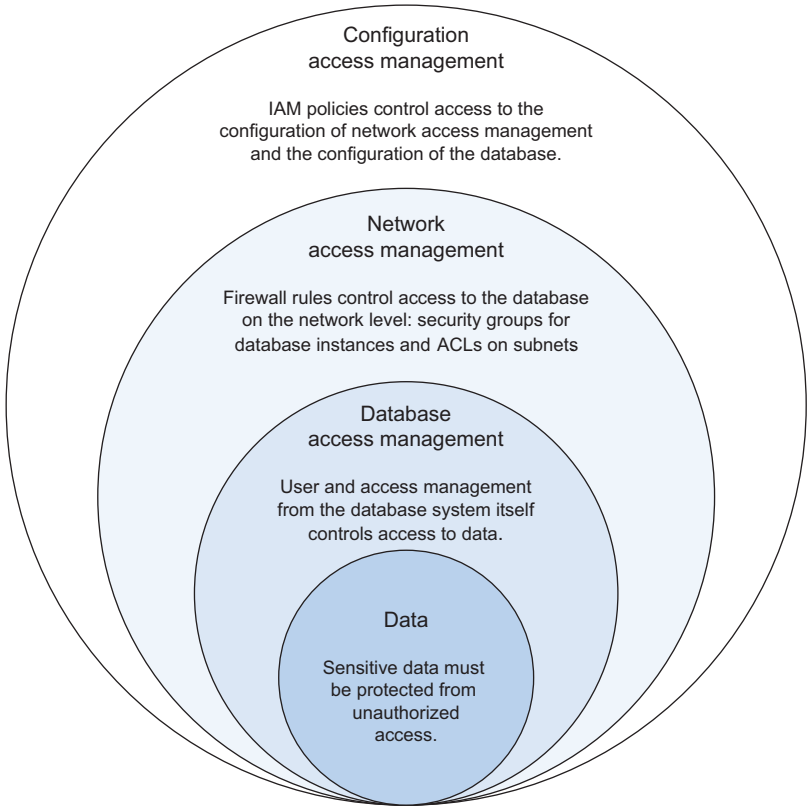


Figure 10.4 Your data is protected by the database itself, security groups, and IAM.

10.4.1 Controlling access to the configuration of an RDS database

Access to the RDS service is controlled using the IAM service. IAM is responsible for controlling access to actions like creating, updating, and deleting an RDS database instance. IAM doesn't manage access inside the database; that's the job of the database engine. IAM policies define which configuration and management actions an identity is allowed to execute on RDS. You attach these policies to IAM users, groups, or roles to control what actions they can perform on the database.

The following listing shows an IAM policy that allows access to all RDS configuration and management actions. You could use this policy to limit access by attaching it only to trusted IAM users and groups.

Listing 10.4 Allowing access to all RDS service configuration and management actions

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

Allows the specified actions on the specified resources

All possible actions on RDS service are specified (e.g., changes to the database configuration).

All RDS databases are specified.

Only people and machines that really need to make changes to RDS databases should be allowed to do so. The following listing shows an IAM policy that denies all destructive actions to prevent data loss by human failure.

Listing 10.5 IAM policy denying destructive actions

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

Allows access …

… to all actions related to RDS …

… and all resources.

But, denies access …

… to all destructive actions on the RDS service (e.g., delete database instance) …

… for all resources.

As discussed in chapter 5, when introducing IAM, a `Deny` statement overrides any `Allow` statement. Therefore, a user or role with the IAM policy attached does have limited access to RDS, because all actions are allowed except the ones that are destructive.

10.4.2 Controlling network access to an RDS database

An RDS database is linked to security groups. Each security group consists of rules for a firewall controlling inbound and outbound database traffic. You already know about using security groups in combination with virtual machines.

The next listing shows the configuration of the security group attached to the RDS database in our WordPress example. Inbound connections to port 3306 (the default port for MySQL) are allowed only from virtual machines linked to the security group called `WebServerSecurityGroup`.

Listing 10.6 CloudFormation template extract: Firewall rules for an RDS database

```
DatabaseSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'awsinaction-db-sg'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - IpProtocol: tcp
      FromPort: 3306
      ToPort: 3306
      SourceSecurityGroupId: !Ref WebServerSecurityGroup
```

The security group for the database instance, allowing incoming traffic on the MySQL default port for web servers

The default MySQL port is 3306.

References the security group for web servers

Only machines that really need to connect to the RDS database should be allowed to do so on the network level, such as EC2 instances running your web server or application server. See chapter 5 if you're interested in more details about security groups (firewall rules).

10.4.3 Controlling data access

A database engine also implements access control itself. User management of the database engine has nothing to do with IAM users and access rights; it's only responsible for controlling access to the database. For example, you typically define a user for each application and grant rights to access and manipulate tables as needed. In the WordPress example, a database user called wordpress is created. The WordPress application authenticates itself to the database engine (MySQL, in this case) with this database user and a password.

IAM database authentication

AWS provides an IAM database authentication mechanism for MariaDB, MySQL, and PostgreSQL. With IAM database authentication, you no longer need to create users with a username and password in the database engine. Instead, you create a database user that uses a plug-in called `AWSAuthenticationPlugin` for authentication. You then log in to the database with the username and a token that is generated with your IAM identity. The token is valid for 15 minutes, so you have to renew it from time to time. You can learn more about IAM database authentication in the AWS documentation at http://mng.bz/z57r.

Typical use cases follow:

- Limiting write access to a few database users (e.g., only for an application)
- Limiting access to specific tables to a few users (e.g., to one department in the organization)
- Limiting access to tables to isolate different applications (e.g., hosting multiple applications for different customers on the same database)

User and access management varies between database systems. We don't cover this topic in this book; refer to your database system's documentation for details.

10.5 Building on a highly available database

The availability of our blog cloudonaut.io is key to our business success. That's the case for your WordPress site as well. Therefore, you should avoid downtimes when possible. This chapter is about increasing the availability of your database.

The database is typically the most important part of a system. Applications won't work if they can't connect to the database, and the data stored in the database is mission critical, so the database must be highly available and store data durably.

Amazon RDS lets you launch highly available (HA) databases. Compared to a default database consisting of a single database instance, an HA RDS database consists of two database instances: a primary and a standby database instance. You will be paying for both instances. All clients send requests to the primary database. Data is replicated between the primary and the standby database synchronously, as shown in figure 10.5.

We strongly recommend using high-availability deployment for all databases that handle production workloads. If you want to save money, you can choose not to deploy a highly available database for your test systems.

If the primary database becomes unavailable due to hardware or network failures, RDS starts the failover process. The standby database then becomes the primary database. As figure 10.6 shows, the DNS name is updated and clients begin to use the former standby database for their requests.

RDS detects the need for a failover automatically and executes it without human intervention.

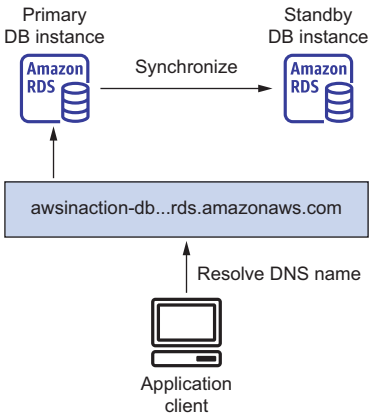


Figure 10.5 The primary database is replicated to the standby database when running in high-availability mode.

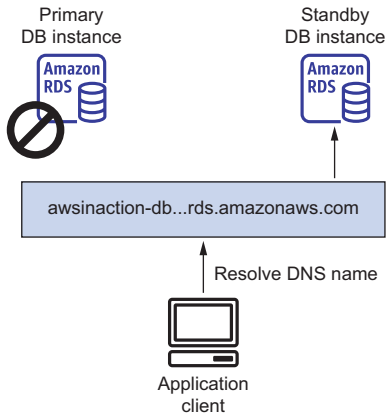


Figure 10.6 The client fails over to the standby database if the primary database fails, using DNS resolution.

Aurora is different

Aurora is an exception to the way that highly available databases operate in AWS. It does not store your data on a single EBS volume. Instead, Aurora stores data on a cluster volume. A cluster volume consists of multiple disks, with each disk having a copy of the cluster data. This implies that the storage layer of Aurora is not a single point of failure. But still, only the primary Aurora database instance accepts write requests. If the primary goes down, it is automatically re-created, which typically takes less than 10 minutes. If you have replica instances in your Aurora cluster, a replica is promoted to be the new primary instance, which usually takes around one minute and is much faster than primary re-creation.

Multi-AZ with two standby instances

AWS introduced a new option for multi-AZ deployments of RDS databases: Multi-AZ with two standby instances. The advantages compared to the single standby instance that we discussed follow:

1. Both standby instances can be used as read replicas to increase capacity for read-only queries. You will learn more about read replicas later.
2. Lower latency and jitter for transaction commits, which improves the write performance.
3. Faster failover in less than 60 seconds.

Right now, this option is available only for PostgreSQL and MySQL engines. Check out https://aws.amazon.com/rds/features/multi-az/ to learn more.

10.5.1 Enabling high-availability deployment for an RDS database

WARNING Starting a highly available RDS database will incur charges, about USD $0.017000 per hour. See https://aws.amazon.com/rds/pricing/ if you want to find out the current hourly price.

Execute the following command at your local terminal to enable high-availability deployment for the RDS database you started in section 10.1:

```
$ aws cloudformation update-stack --stack-name wordpress --template-url \
➥ https://s3.amazonaws.com/awsinaction-code3/\
➥ chapter10/template-multiaz.yaml \
➥ --parameters ParameterKey=WordpressAdminPassword,UsePreviousValue=true \
➥ --capabilities CAPABILITY_IAM
```

The RDS database is updated based on a slightly modified CloudFormation template as shown in the next listing.

Listing 10.7 Modifying the RDS database by enabling high availability

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.t2.micro'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
    MultiAZ: true
  DependsOn: VPCGatewayAttachment
```

Enables high-availability deployment for the RDS database

It will take several minutes for the database to be deployed in HA mode. There is nothing more you need to do—the database is now highly available.

What is Multi-AZ?

Each AWS region is split into multiple independent data centers, which are also called availability zones. We introduced the concept of availability zones in chapters 4 and 5, but skipped one aspect of HA deployment that is used only for RDS: the primary and standby databases are launched into two different availability zones. AWS calls the high-availability deployment of RDS Multi-AZ deployment for this reason.

In addition to the fact that a high-availability deployment increases your database's reliability, it offers another important advantage: reconfiguring or maintaining a single-mode database causes short downtimes. High-availability deployment of an RDS database solves this problem because AWS switches to the standby database during maintenance.

10.6 Tweaking database performance

When search engines decide in which order to present the search results, the loading speed of a website is an important factor. Therefore, it is important to optimize the performance of your WordPress site. You will learn how to make sure the MySQL database is not slowing down your website in this section.

The easiest way to scale a RDS database, or an SQL database in general, is to scale *vertically*. Scaling a database vertically means increasing the following resources of your database instance:

- Faster CPU
- More memory
- Faster storage

Keep in mind that you can't scale vertically (which means increasing resources) without limits. One of the largest RDS database instance types comes with 32 cores and 244 GiB memory. In comparison, an object store like S3 or a NoSQL database like DynamoDB can be scaled horizontally without limits, because they add more machines to the cluster if additional resources are needed.

10.6.1 Increasing database resources

When you start an RDS database, you choose an instance type. The instance type defines the computing power and memory of your virtual machine (such as when you start an EC2 instance). Choosing a bigger instance type increases computing power and memory for RDS databases.

You started an RDS database with instance type `db.t2.micro`, the smallest available instance type. You can change the instance type using a CloudFormation template, the CLI, the Management Console, or AWS SDKs. You may want to increase the instance type if performance is inadequate. You will learn how to measure performance in section 10.7. Listing 10.8 shows how to change the CloudFormation template to increase the instance type from `db.t2.micro` with one virtual core and 615 MB memory to `db.m3.large` with two faster virtual cores and 7.5 GB memory. You'll do this only in theory. Don't do this to your running database because it is not covered by the Free Tier and will incur charges. Keep in mind that modifying the instance type causes a short downtime.

Listing 10.8 Modifying the instance type to improve performance of an RDS database

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.m3.large'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
    MultiAZ: true
  DependsOn: VPCGatewayAttachment
```

Increases the size of the underlying virtual machine for the database instance from db.t2.micro to db.m3.large

Because a database has to read and write data to a disk, I/O performance is important for the database's overall performance. RDS offers the following three different types of storage, as you already know from reading about the block storage service EBS:

- General purpose (SSD)
- Provisioned IOPS (SSD)
- Magnetic

You should choose general-purpose (SSD) or even provisioned IOPS (SSD) storage for production workloads. The options are exactly the same as when using EBS for virtual machines. If you need to guarantee a high level of read or write throughput, you should use provisioned IOPS (SSD). The general-purpose (SSD) option offers moderate baseline performance with the ability to burst. The throughput for general purpose (SSD) depends on the amount of initialized storage size. Magnetic storage is an option if you need to store data at a low cost, or if you don't need to access it in a predictable, performant way. The next listing shows how to enable general-purpose (SSD) storage using a CloudFormation template.

Listing 10.9 Modifying the storage type to improve performance of an RDS database

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.m3.large'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
    MultiAZ: true
    StorageType: 'gp2'
  DependsOn: VPCGatewayAttachment
```

Uses general-purpose (SSD) storage to increase I/O performance

10.6.2 Using read replication to increase read performance

A database suffering from too many read requests can be scaled horizontally by adding additional database instances for read traffic and enabling replication from the primary (writable) copy of the database instance. As figure 10.7 shows, changes to the database are asynchronously replicated to an additional read-only database instance. The read requests can be distributed between the primary database and its read-replication databases to increase read throughput. Be aware that you need to implement the distinction between read and write requests on the application level.

Tweaking read performance with replication makes sense only if the application generates many read requests and few write requests. Fortunately, most applications read more than they write.

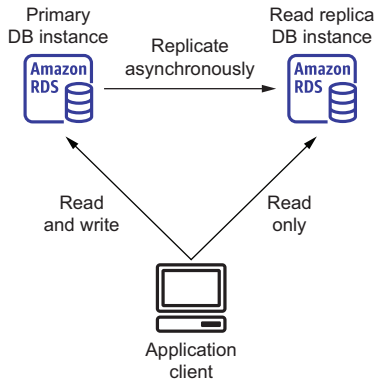


Figure 10.7 Read requests are distributed between the primary and read-replication databases for higher read performance.

CREATING A READ-REPLICATION DATABASE

Amazon RDS supports read replication for MySQL, MariaDB, PostgreSQL, Oracle, and SQL Server databases. To use read replication, you need to enable automatic backups for your database, as shown in section 10.3.

> **WARNING** Starting an RDS read replica will incur charges. See https://aws.amazon.com/rds/pricing/ if you want to find out the current hourly price.

Execute the following command from your local machine to create a read-replication database for the WordPress database you started in section 10.1. Replace the `$DBInstanceIdentifier` with the value from `aws rds describe-db-instances --query "DBInstances[0].DBInstanceIdentifier" --output text`:

```
$ aws rds create-db-instance-read-replica \
➥ --db-instance-identifier awsinaction-db-read \
➥ --source-db-instance-identifier $DBInstanceIdentifier
```

RDS automatically triggers the following steps in the background:

1. Creating a snapshot from the source database, also called the primary database instance
2. Launching a new database based on that snapshot
3. Activating replication between the primary and read-replication database instances
4. Creating an endpoint for SQL read requests to the read-replication database instances

After the read-replication database is successfully created, it's available to answer SQL read requests. The application using the SQL database must support the use of read-replication databases. WordPress, for example, doesn't support read replicas by default, but you can use a plug-in called HyperDB to do so; the configuration is tricky, so we'll skip this part. You can get more information here: https://wordpress.org/plugins/hyperdb/.

Creating or deleting a read replica doesn't affect the availability of the primary (writable) database instance.

Using read replication to transfer data to another region

RDS supports read replication between regions for Aurora, MariaDB, MySQL, Oracle, and PostgreSQL databases. You can replicate your data from the data centers in North Virginia to the data centers in Ireland, for example. Three major use cases for this feature follow:

- Backing up data to another region for the unlikely event of an outage covering a complete region
- Transferring data to another region to be able to answer read requests with lower latency
- Migrating a database to another region

Creating read replication between two regions incurs an additional cost because you have to pay for the transferred data.

PROMOTING A READ REPLICA TO A STANDALONE DATABASE

Imagine your WordPress site became really popular. On the one hand, that's a great success. On the other hand, you are struggling with handling all that load with your cloud infrastructure. You are thinking about adding read replicas to your database to decrease the load on the primary database.

If you create a read-replication database to migrate a database from one region to another, or if you have to perform heavy and load-intensive tasks on your database, such as adding an index, it's helpful to switch your workload from the primary database to a read-replication database. The read replica must become the new primary database.

The following command promotes the read-replica database you created in this section to a standalone primary database. Note that the read-replication database will perform a restart and be unavailable for a few minutes:

```
$ aws rds promote-read-replica --db-instance-identifier awsinaction-db-read
```

The RDS database instance named awsinaction-db-read will accept write requests after the transformation is successful.

Cleaning up

It's time to clean up, to avoid unwanted expense. Execute the following command:

```
$ aws rds delete-db-instance --db-instance-identifier \
➥ awsinaction-db-read --skip-final-snapshot
```

You've gained experience with the AWS relational database service in this chapter. We'll end the chapter by taking a closer look at the monitoring capabilities of RDS.

10.7 Monitoring a database

To avoid downtime of your WordPress site, it is imperative that you monitor all important parts of your cloud infrastructure. The database is definitely one of the key components. That's why you will learn about RDS monitoring in this section.

RDS is a managed service. Nevertheless, you need to monitor some metrics yourself to make sure your database can respond to all requests from applications. RDS publishes several metrics for free to AWS CloudWatch, a monitoring service for the AWS cloud. You can watch these metrics through the Management Console, as shown in figure 10.8, and define alarms for when a metric reaches a threshold.

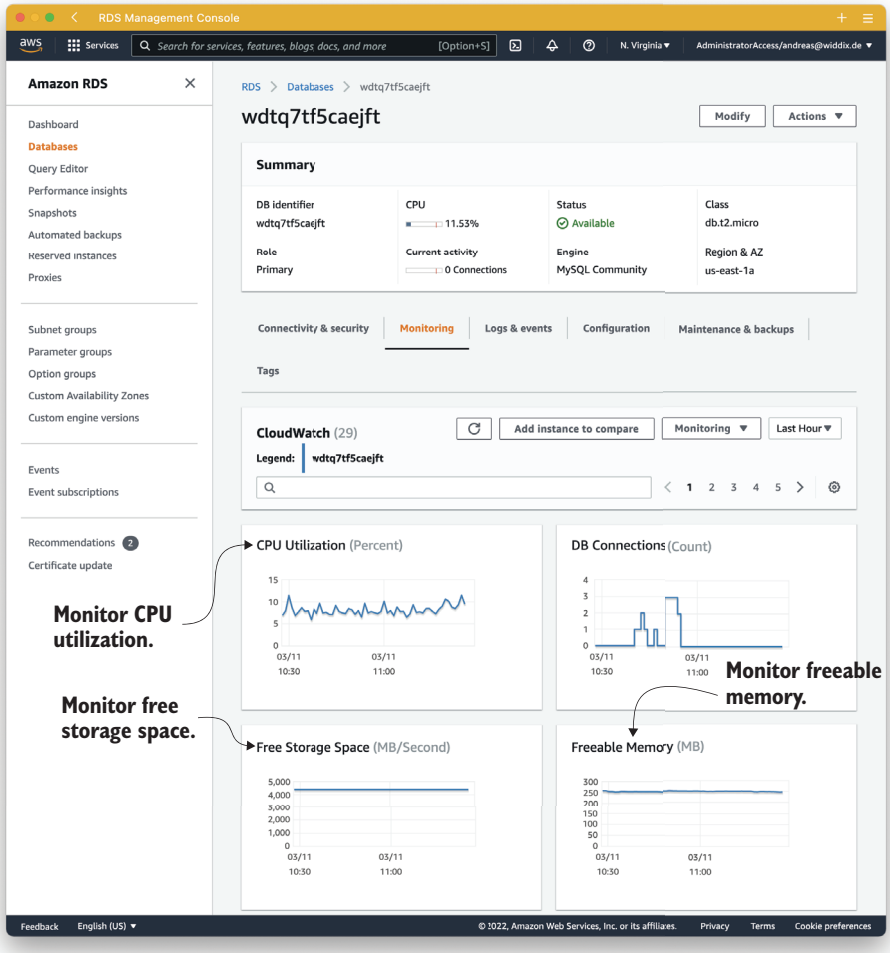


Figure 10.8 Metrics to monitor an RDS database from the Management Console

Table 10.4 shows the most important metrics; we recommend that you monitor them by creating alarms.

Table 10.4 Important metrics for RDS databases from CloudWatch

Name	Description
`FreeStorageSpace`	Available storage in bytes. Make sure you don't run out of storage space. We recommend setting the alarm threshold to < 2147483648 (2 GB).
`CPUUtilization`	The usage of the CPU as a percentage. High utilization can be an indicator of a bottleneck due to insufficient CPU performance. We recommend setting the alarm threshold to > 80%.
`FreeableMemory`	Free memory in bytes. Running out of memory can cause performance problems. We recommend setting the alarm threshold to < 67108864 (64 MB).
`DiskQueueDepth`	Number of outstanding requests to the disk. A long queue indicates that the database has reached the storage's maximum I/O performance. We recommend setting the alarm threshold to > 64.
`SwapUsage`	If the database has insufficient memory, the OS starts to use the disk as memory (this is called swapping). Using the disk as memory is slow and will cause performance problems. We recommend setting the alarm threshold to > 268435456 (256 MB).

We recommend that you monitor these metrics in particular, to make sure your database isn't the cause of application performance problems.

Cleaning up

It's time to clean up to avoid unwanted expense. Execute the following command to delete all resources corresponding to the WordPress blogging platform based on an RDS database:

```
$ aws cloudformation delete-stack --stack-name wordpress
```

In this chapter, you've learned how to use the RDS service to manage relational databases for your applications. The next chapter will focus on in-memory caches.

Summary

- RDS is a managed service that provides relational databases.
- You can choose between PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server databases. Aurora is the database engine built by Amazon.
- The fastest way to import data into an RDS database is to copy it to a virtual machine in the same region and load it into the RDS database from there.
- RDS comes with built-in backup and restore functionality allowing you to create and restore snapshots on demand as well as to restore a database to a certain point in time.

- You can control access to data with a combination of IAM policies and firewall rules and on the database level.
- You can restore an RDS database to any time in the retention period (a maximum of 35 days).
- RDS databases can be highly available. You should launch RDS databases in Multi-AZ mode for production workloads.
- Read replication can improve the performance of read-intensive workloads on an SQL database.
- CloudWatch metrics allow you to monitor your database, for example, to debug performance problems or decide when to increase or decrease the size of your database instance.

11 Caching data in memory: Amazon ElastiCache and MemoryDB

This chapter covers

- The benefits of a having a caching layer between your application and data store
- Defining key terminology, such as cache cluster, node, shard, replication group, and node group
- Using/operating an in-memory key-value store
- Performance tweaking and monitoring ElastiCache clusters

Imagine a relational database being used for a popular mobile game where players' scores and ranks are updated and read frequently. The read and write pressure to the database will be extremely high, especially when ranking scores across millions of players. Mitigating that pressure by scaling the database may help with load but not necessarily the latency or cost. Also, caching relational databases tends to be more expensive than caching data stores.

A proven solution used by many gaming companies is leveraging an in-memory data store such as Redis for both caching and ranking player and game metadata. Instead of reading and sorting the leaderboard directly from the relational database, they store an in-memory game leaderboard in Redis, commonly using a sorted set, which will sort the data automatically when it's inserted, based on the

score parameter. The score value may consist of the actual player ranking or player score in the game.

Because the data resides in memory and does not require heavy computation to sort, retrieving the information is incredibly fast, leaving little reason to query directly from the relational database. In addition, any other game and player metadata, such as player profile, game-level information, and so on, that requires heavy reads can also be cached using this in-memory layer, thus alleviating heavy read traffic to and from the database.

In this solution, both the relational database and in-memory layer will store updates to the leaderboard: one will serve as the primary database and the other as the working and fast processing layer. When implementing a caching layer, you can employ a variety of caching techniques to keep the data that's cached fresh, which we'll review later. Figure 11.1 shows that the cache sits between your application and the database.

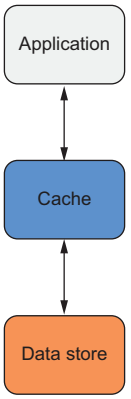


Figure 11.1 The cache sits between the application and the database.

A cache comes with the following benefits:

- Serving read traffic from the caching layer frees resources on your primary data store, for example, for write requests.
- It speeds up your application because the caching layer responds more quickly than your data store.
- It allows you to downsize your data store, which is typically more expensive than the caching layer.

Most caching layers reside in memory, which is why they are so fast. The downside is that you can lose the cached data at any time because of a hardware defect or a restart. Always keep a copy of your data in a primary data store with disk durability, like the relational database in the mobile game example. Alternatively, Redis has optional failover support. In the event of a node failure, a replica node will be elected to be the new primary and will already have a copy of the data. On top of that, some in-memory databases also support writing data to persistent storage to be able to restore the data after a reboot or outage.

In this chapter, you will learn how to implement an in-memory caching layer to improve the performance of a web application. You will deploy a complex web application called Discourse, a modern forum software application that uses Redis for caching. You will also learn how to scale a cache cluster and how to monitor and tweak performance.

Depending on your caching strategy, you can either populate the cache in real time or on demand. In the mobile game example, on demand means that if the leaderboard is not in the cache, the application asks the relational database and puts the result into the cache. Any subsequent request to the cache will result in a cache hit,

meaning the data is found in the cache. This will be true until the duration of the TTL (time-to-live) value on the cached value expires. This strategy is called *lazy-loading* data from the primary data store. Additionally, we could have a job running in the background that queries the leaderboard from the relational database every minute and puts the result in the cache to populate the cache in advance.

The lazy-loading strategy (getting data on demand) is implemented like this:

1. The application writes data to the data store.
2. If the application wants to read the data, at a later time it makes a request to the caching layer.
3. If the caching layer does not contain the data, the application reads from the data store directly and puts the value into the cache, and also returns the value to the client.
4. Later, if the application wants to read the data again, it makes a request to the caching layer and finds the value.

This strategy comes with a problem. What if the data is changed while it is in the cache? The cache will still contain the old value. That's why setting an appropriate TTL value is critical to ensure cache validity. Let's say you apply a TTL of five minutes to your cached data: this means you accept that the data could be out of sync by up to five minutes with your primary database. Understanding the frequency of change for the underlying data and the effects out-of-sync data will have on the user experience is the first step of identifying the appropriate TTL value to apply. A common mistake some developers make is assuming that a few seconds of a cache TTL means that having a cache is not worthwhile. Remember that within those few seconds, millions of requests can be offloaded from your data store, speeding up your application and reducing database pressure. Performance testing your application with and without your cache, along with various caching approaches, will help fine-tune your implementation. In summary, the shorter the TTL, the higher the load on the underlying data store. The higher the TTL, the more out of sync the data gets.

The write-through strategy (caching data up front) is implemented differently to tackle the synchronization problem, as shown here:

1. The application writes data to the data store and the cache (or the cache is filled asynchronously, for example, by a background job, an AWS Lambda function, or the application).
2. If the application wants to read the data at a later time, it makes a request to the caching layer, which always contains the latest data.
3. The value is returned to the client.

This strategy also comes with a problem. What if the cache is not big enough to contain all your data? Caches are in memory, and your data store's disk capacity is usually larger than your cache's memory capacity. When your cache exceeds the available memory, it will evict data or stop accepting new data. In both situations, the application stops

working. A global leaderboard will most likely fit into the cache. Imagine that a leaderboard is 4 KB in size and the cache has a capacity of 1 GB (1,048,576 KB). But what about introducing leaderboards per team? You can only store 262,144 (1,048,576/4) leaderboards, so if you have more teams than that, you will run into a capacity issue. Figure 11.2 compares the two caching strategies.

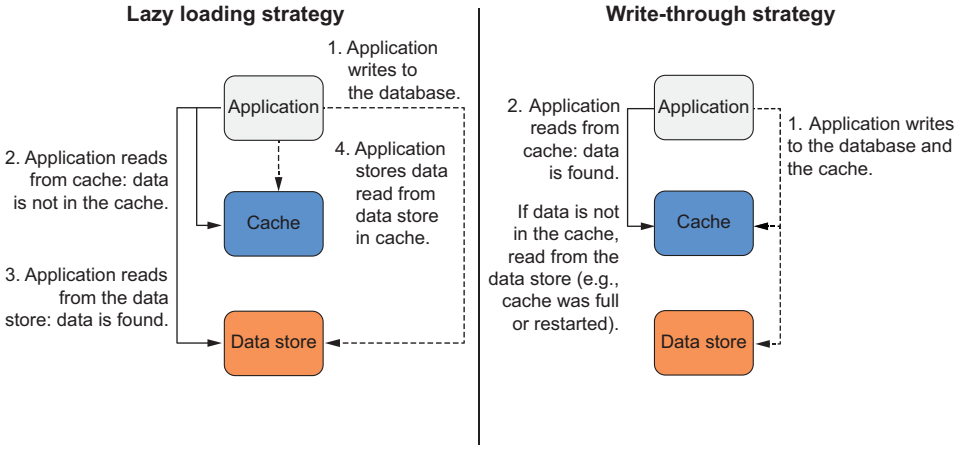


Figure 11.2 Comparing the lazy-loading and write-through caching strategies

When evicting data, the cache needs to decide which data it should delete. One popular strategy is to evict the least recently used (LRU) data. This means that cached data must contain meta information about the time when it was last accessed. In case of an LRU eviction, the data with the oldest timestamp is chosen for eviction.

Caches are usually implemented using key-value stores. Key-value stores don't support sophisticated query languages such as SQL. They support retrieving data based on a key, usually a string, or specialized commands, for example, to extract sorted data efficiently.

Imagine that in your relational database is a `player` table for your mobile game. One of the most common queries is `SELECT id, nickname FROM player ORDER BY score DESC LIMIT 10` to retrieve the top 10 players. Luckily, the game is very popular, but this comes with a technical challenge. If many players look at the leaderboard, the database becomes very busy, which causes high latency or even timeouts. You have to come up with a plan to reduce the load on the database. As you have already learned, caching can help. What technique should you employ for caching? You have a few options.

One approach you can take with Redis is to store the result of your SQL query as a string value and the SQL statement as your key name. Instead of using the whole SQL query as the key, you can hash the string with a hash function like `md5` or `sha256` to

optimize storage and bandwidth, shown in figure 11.3. Before the application sends the query to the database, it takes the SQL query as the key to ask the caching layer for data—step 2. If the cache does not contain data for the key (step 3), the SQL query is sent to the relational database (step 4). The result (step 5) is then stored in the cache using the SQL query as the key (step 6). The next time the application wants to perform the query, it asks the caching layer (step 7), which now contains the cached table (step 8).

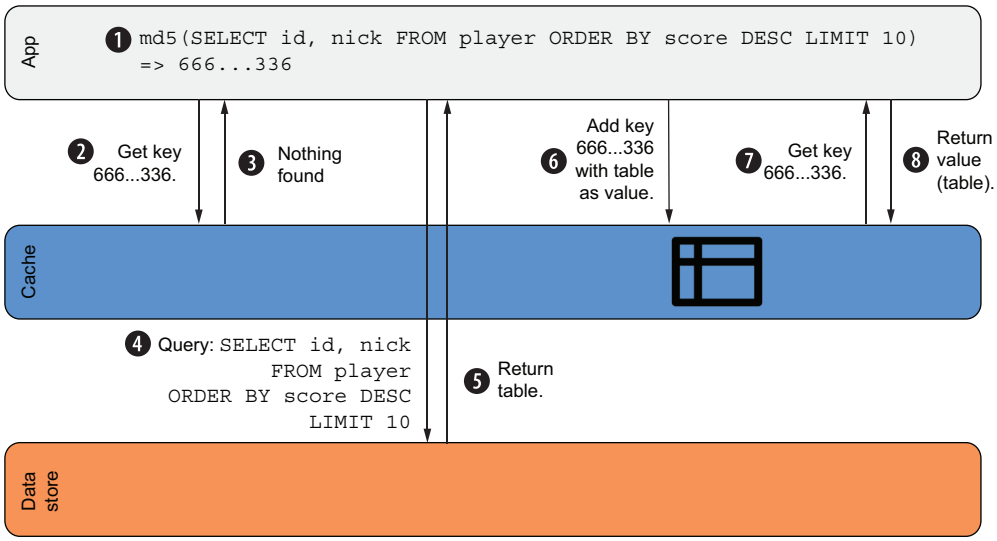


Figure 11.3 SQL caching layer implementation

To implement caching, you only need to know the key of the cached item. This can be an SQL query, a filename, a URL, or a user ID. You take the key and ask the cache for a result. If no result is found, you make a second call to the underlying data store, which knows the truth.

With Redis, you also have the option of storing the data in other data structures such as a sorted set. If the data is stored in a sorted set, retrieving the ranked data will be very efficient. You could simply store players and their scores and sort by the score. An equivalent SQL command would be as follows:

```
ZREVRANGE "player-scores" 0 9
```

This would return the 10 players in a sorted set named "player-scores," ordered from highest to lowest.

The two most popular implementations of in-memory key-value stores are Memcached and Redis. Table 11.1 compares their features.

Table 11.1 Comparing Memcached and Redis features

	Memcached	Redis
Data types	Simple	Complex
Data manipulation commands	12	125
Server-side scripting	No	Yes (Lua)
Transactions	No	Yes

> **Examples are 100% covered by the Free Tier**
> The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end of the sections.

Amazon ElastiCache offers Memcached and Redis clusters as a service. Therefore, AWS covers the following aspects for you:

- *Installation*—AWS installs the software for you and has enhanced the underlying engines.
- *Administration*—AWS administers Memcached/Redis for you and provides ways to configure your cluster through parameter groups. AWS also detects and automates failovers (Redis only).
- *Monitoring*—AWS publishes metrics to CloudWatch for you.
- *Patching*—AWS performs security upgrades in a customizable time window.
- *Backups*—AWS optionally backs up your data in a customizable time window (Redis only).
- *Replication*—AWS optionally sets up replication (Redis only).

Next, you will learn how to create an in-memory cluster with ElastiCache that you will later use as an in-memory cache for a gaming backend.

11.1 Creating a cache cluster

In this chapter, we focus on the Redis engine because it's more flexible. You can choose which engine to use based on the features that we compared in the previous section. If there are significant differences to Memcached, we will highlight them.

11.1.1 Minimal CloudFormation template

Imagine you are developing an online game. To do so, you need to store sessions that keep the current game state of each user, as well as a highscore list. Latency is important

to ensure the gamers enjoy a great gaming experience. To be able to read and write data with very little latency, you decide to use the in-memory database Redis.

First, you need to create an ElastiCache cluster using the Management Console, the CLI, or CloudFormation. You will use CloudFormation in this chapter to manage your cluster. The resource type of an ElastiCache cluster is `AWS::ElastiCache::CacheCluster`. The required properties follow:

- `Engine`—Either `redis` or `memcached`
- `CacheNodeType`—Similar to the EC2 instance type, for example, `cache.t2.micro`
- `NumCacheNodes`—1 for a single-node cluster
- `CacheSubnetGroupName`—Reference subnets of a VPC using a dedicated resource called a subnet group
- `VpcSecurityGroupIds`—The security groups you want to attach to the cluster

A minimal CloudFormation template is shown in the next listing. The first part of the template contains the ElastiCache cluster.

Listing 11.1 Minimal CloudFormation template of an ElastiCache Redis single-node cluster

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 11 (minimal)'
Parameters:
  VPC:
    Type: 'AWS::EC2::VPC::Id'
  SubnetA:
    Type: 'AWS::EC2::Subnet::Id'
  SubnetB:
    Type: 'AWS::EC2::Subnet::Id'
Resources:
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        CidrIp: '0.0.0.0/0'
  CacheSubnetGroup:
    Type: 'AWS::ElastiCache::SubnetGroup'
    Properties:
      Description: cache
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
  Cache:
    Type: 'AWS::ElastiCache::CacheCluster'
```

Defines VPC and subnets as parameters

The security group to manage which traffic is allowed to enter/leave the cluster

Redis listens on port 6379. This allows access from all IP addresses, but because the cluster has only private IP addresses, access is possible only from inside the VPC. You will improve this in section 11.3.

Subnets are defined within a subnet group (same approach is used in RDS).

List of subnets that can be used by the cluster

The resource to define the Redis cluster

```
    Properties:
      CacheNodeType: 'cache.t2.micro'
      CacheSubnetGroupName: !Ref CacheSubnetGroup
      Engine: redis
      NumCacheNodes: 1
      VpcSecurityGroupIds:
      - !Ref CacheSecurityGroup
```

The node type cache.t2.micro comes with 0.555 GiB memory and is part of the Free Tier.

ElastiCache supports redis and memcached. We are using redis because we want to make use of advanced data structures supported only by Redis.

Creates a single-node cluster for testing, which is not recommended for production workloads because it introduces a single point of failure

As mentioned already, ElastiCache nodes in a cluster use only private IP addresses. Therefore, you can't connect to a node directly over the internet. The same is true for other resources as RDS database instances. Therefore, create an EC2 instance in the same VPC as the cluster for testing. From the EC2 instance, you can then connect to the private IP address of the cluster.

11.1.2 Test the Redis cluster

To be able to access the ElastiCache cluster, you'll need a virtual machine. The following snippet shows the required resources:

```
Resources:
  # [...]
  InstanceSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'vm'
      VpcId: !Ref VPC
  Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      InstanceType: 't2.micro'
      IamInstanceProfile: !Ref InstanceProfile
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref InstanceSecurityGroup
        SubnetId: !Ref SubnetA
Outputs:
  InstanceId:
    Value: !Ref Instance
    Description: 'EC2 Instance ID (connect via Session Manager)'
  CacheAddress:
    Value: !GetAtt 'Cache.RedisEndpoint.Address'
    Description: 'Redis DNS name (resolves to a private IP address)'
```

The security group does allow outbound traffic only.

The virtual machine used to connect to your Redis cluster

The ID of the instance used to connect via the Session Manager

The DNS name of Redis cluster node (resolves to a private IP address)

Next, create a stack based on the template to create all the resources in your AWS account using the Management Console: http://mng.bz/09jm. You have to fill in the next three parameters when creating the stack:

- `SubnetA`—You should have at least two options here; select the first one.
- `SubnetB`—You should have at least two options here; select the second one.
- `VPC`—You should have only one possible VPC here—your default VPC. Select it.

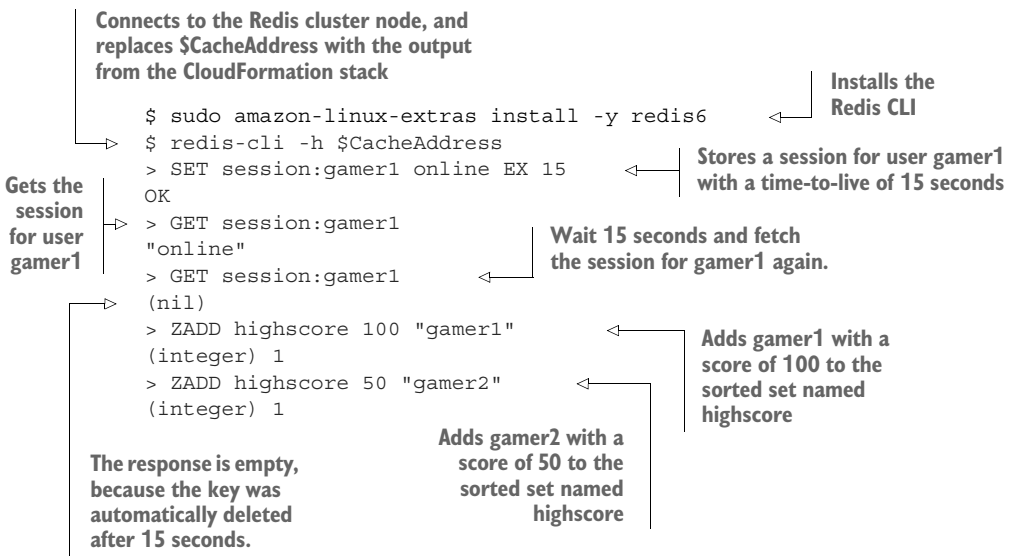
Want to have a deeper look into the CloudFormation template?

Where is the template located?

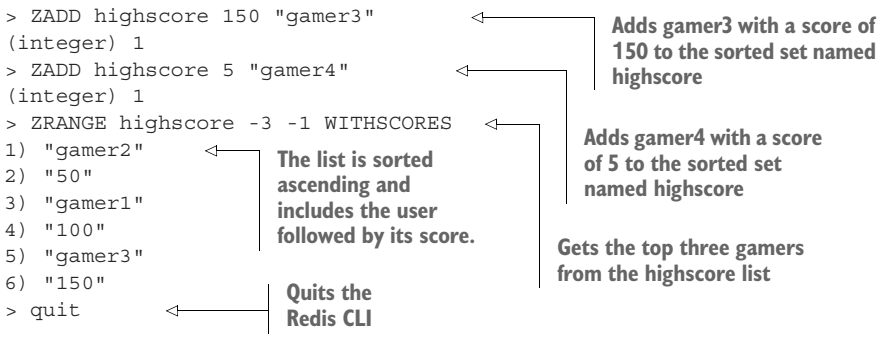
You can find the template on GitHub. Download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter11/redis-minimal.yaml. On S3, the same file is located at http://mng.bz/9Vra.

The following example guides you through storing a gamer's session, as well as populating and reading a highscore list:

1. Wait until the CloudFormation stack reaches status `CREATE_COMPLETE`.
2. Select the stack and open the Outputs tab. Copy the EC2 instance ID and cache address.
3. Use the Session Manager to connect to the EC2 instance, and use the Redis CLI to interact with the Redis cluster node as described in the following listing.
4. Execute the following commands. Don't forget to replace `$CacheAddress` with the value from the CloudFormation outputs:




```
$ sudo amazon-linux-extras install -y redis6
$ redis-cli -h $CacheAddress
> SET session:gamer1 online EX 15
OK
> GET session:gamer1
"online"
> GET session:gamer1
(nil)
> ZADD highscore 100 "gamer1"
(integer) 1
> ZADD highscore 50 "gamer2"
(integer) 1
```



You've successfully connected to the Redis cluster and simulated a session store and highscore list used by a typical gaming backend. With this knowledge, you could start to implement a caching layer in your own application. But as always, there are more options to discover.

Cleaning up

It's time to delete the CloudFormation stack named `redis-minimal`. Use the AWS Management Console or the following command to do so:

```
$ aws cloudformation delete-stack --stack-name redis-minimal
```

Continue with the next section to learn more about advanced deployment options with more than one node to achieve high availability or increase the available memory by using sharding.

11.2 Cache deployment options

Which deployment option you should choose is influenced by the following four factors:

- *Engine*—Which in-memory database do you want to use: Memcached or Redis?
- *Backup/Restore*—Does your workload require data persistence, which means being able to backup and restore data?
- *Replication*—Is high availability important to your workload? If so, you need to replicate to at least one other node.
- *Sharding*—Does your data exceed the maximum memory available for a single node, or do you need to increase the throughput of your system?

Table 11.2 compares the deployment options for Memcached and Redis.

Next, we'll look at deployment options in more detail.

Table 11.2 Comparing ElastiCache and MemoryDB engines and deployment options

Service	Engine	Deployment Option	Backup/Restore	Replication	Sharding
ElastiCache	Memcached	Default	No	No	Yes
	Redis	Single Node	Yes	No	No
		Cluster Mode **disabled**	Yes	Yes	No
		Cluster Mode **enabled**	Yes	Yes	Yes
MemoryDB	Redis	Default	Yes	Yes	Yes

11.2.1 Memcached: Cluster

An Amazon ElastiCache for Memcached cluster consists of 1–40 nodes. Sharding is implemented by the Memcached client, typically using a consistent hashing algorithm, which arranges keys into partitions in a ring distributed across the nodes. The client decides which keys belong to which nodes and directs the requests to those partitions. Each node stores a unique portion of the key-space in memory. If a node fails, the node is replaced, but the data is lost. You cannot back up the data stored in Memcached. Figure 11.4 shows a Memcached cluster deployment. Remember that a VPC is a way to define a private network on AWS. A subnet is a way to separate concerns inside the VPC. The cluster nodes are distributed among multiple subnets to increase availability. The client communicates with the cluster node to get data and write data to the cache.

Use a Memcached cluster if your application requires a simple in-memory store and can tolerate the loss of a node and its data. For instance, the SQL cache example in the beginning of this chapter could be implemented using Memcached. Because the data is always available in the relational database, you can tolerate a node loss, and you need only simple commands (`GET`, `SET`) to implement the query cache.

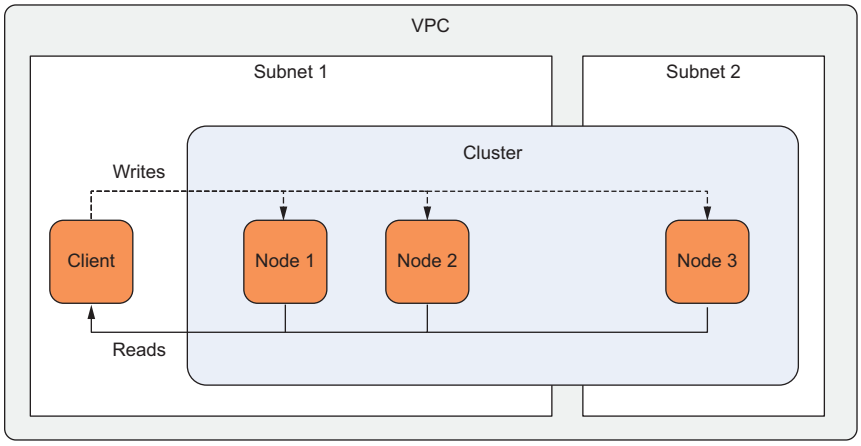


Figure 11.4 Memcached deployment option: Cluster

11.2.2 Redis: Single-node cluster

An ElastiCache for Redis single-node cluster always consists of one node. Sharding and high availability are not possible with a single node. But Redis supports the creation of backups and also allows you to restore those backups. Figure 11.5 shows a Redis single-node cluster.

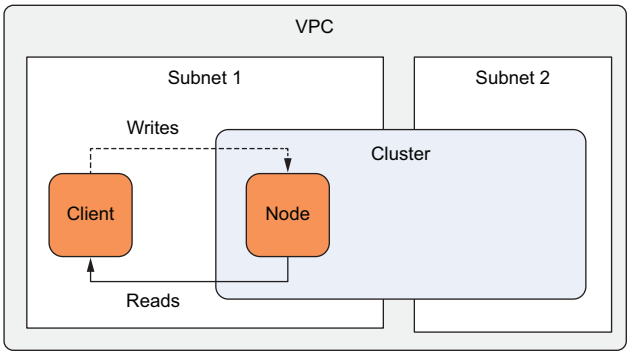


Figure 11.5 Redis deployment option: Single-node cluster

A single node adds a single point of failure (SPOF) to your system. This is probably something you want to avoid for business-critical production systems.

11.2.3 Redis: Cluster with cluster mode disabled

Things become more complicated now, because ElastiCache uses two terminologies. We've been using the terms *cluster*, *node*, and *shard* so far, and the graphical Management Console also uses these terms. But the API, the CLI, and CloudFormation use a different terminology: *replication group*, *node*, and *node group*. We prefer the *cluster*, *node*, and *shard* terminology, but in figures 11.6 and 11.7, we've added the *replication group*, *node*, and *node group* terminology in parentheses.

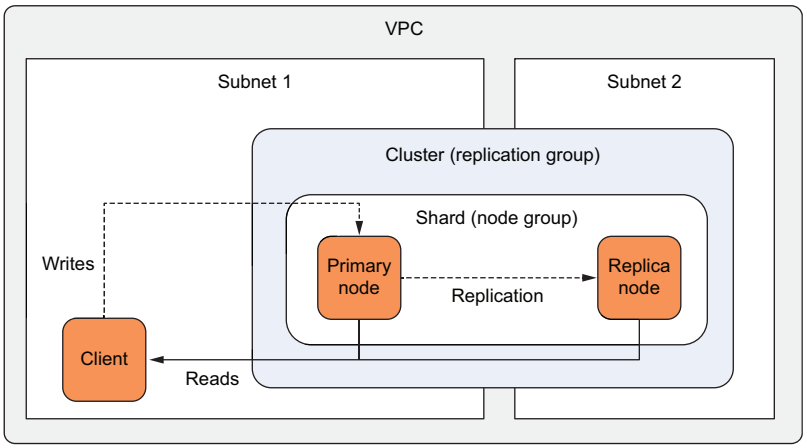


Figure 11.6 Redis deployment option: Cluster with cluster mode disabled

A Redis cluster with cluster mode disabled supports backups and data replication but not sharding. This means there is only one shard containing all the data. The primary node is synchronized to one to five replica nodes.

Use a Redis cluster with cluster mode disabled when you need data replication and all your cached data fits into the memory of a single node. Imagine that your cached data set is 4 GB in size. In that case, the data fits into the memory of nodes of type `cache.m6g.large`, which comes with 6.38 GiB, for example. There is no need to split the data into multiple shards.

11.2.4 Redis: Cluster with cluster mode enabled

A Redis cluster with cluster mode enabled, shown in figure 11.7, supports backups, data replication, and sharding. You can have up to 500 shards per cluster. Each shard consists of at least a primary node and optionally replica nodes. In total, a cluster cannot exceed 500 nodes.

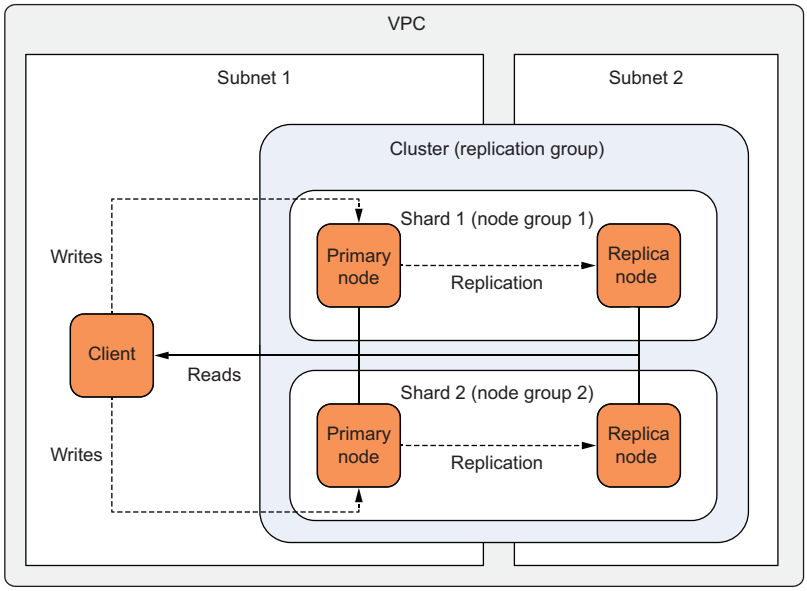


Figure 11.7 Redis deployment option: Cluster with cluster mode enabled

Use a Redis cluster with cluster mode enabled when you need data replication and your data is too large to fit into the memory of a single node. Imagine that your cached data is 22 GB in size. Each cache node has a capacity of 4 GB of memory. Therefore, you will need six shards to get a total capacity of 24 GB of memory. ElastiCache provides up to 437 GB of memory per node, which totals to a maximum cluster capacity of 6.5 TB (15 * 437 GB).

Additional benefits of enabling cluster mode

With cluster mode enabled, failover speed is much faster, because no DNS is involved. Clients are provided a single configuration endpoint to discover changes to the cluster topology, including newly elected primaries. With cluster mode disabled, AWS provides a single primary endpoint, and in the event of a failover, AWS does a DNS swap on that endpoint to one of the available replicas. It may take ~1–1.5 minutes before the application is able to reach the cluster after a failure, whereas with cluster mode enabled, the election takes less than 30 seconds.

On top of that, increasing the number of shards also increases the maximum read/write throughput. If you start with one shard and add a second shard, each shard now only has to deal with 50% of the requests (assuming an even distribution).

Last but not least, as you add nodes, your blast radius decreases. For example, if you have five shards and experience a failover, only 20% of your data is affected. This means you can't write to this portion of the key space until the failover process completes (~15–30 seconds), but you can still read from the cluster, given you have a replica available. With cluster mode disabled, 100% of your data is affected, because a single node consists of your entire key space. You can read from the cluster but can't write until the DNS swap has completed.

You learned about the different deployment options for ElastiCache. There is one more thing we want to bring to your attention: AWS offers another in-memory database called MemoryDB, which you will learn about next.

11.2.5 MemoryDB: Redis with persistence

Even though ElastiCache for Redis supports snapshots and transaction logs for persistence, AWS recommends using ElastiCache as a secondary data store. AWS also provides an alternative called MemoryDB, a proprietary in-memory database with Redis compatibility and distributed transaction log. MemoryDB writes data to disk and reads from memory. By default, MemoryDB stores data among multiple data centers using a distributed transaction log. Therefore, AWS recommends using MemoryDB as a primary database. Keep in mind that data persistence comes with higher write latency—think milliseconds instead of microseconds.

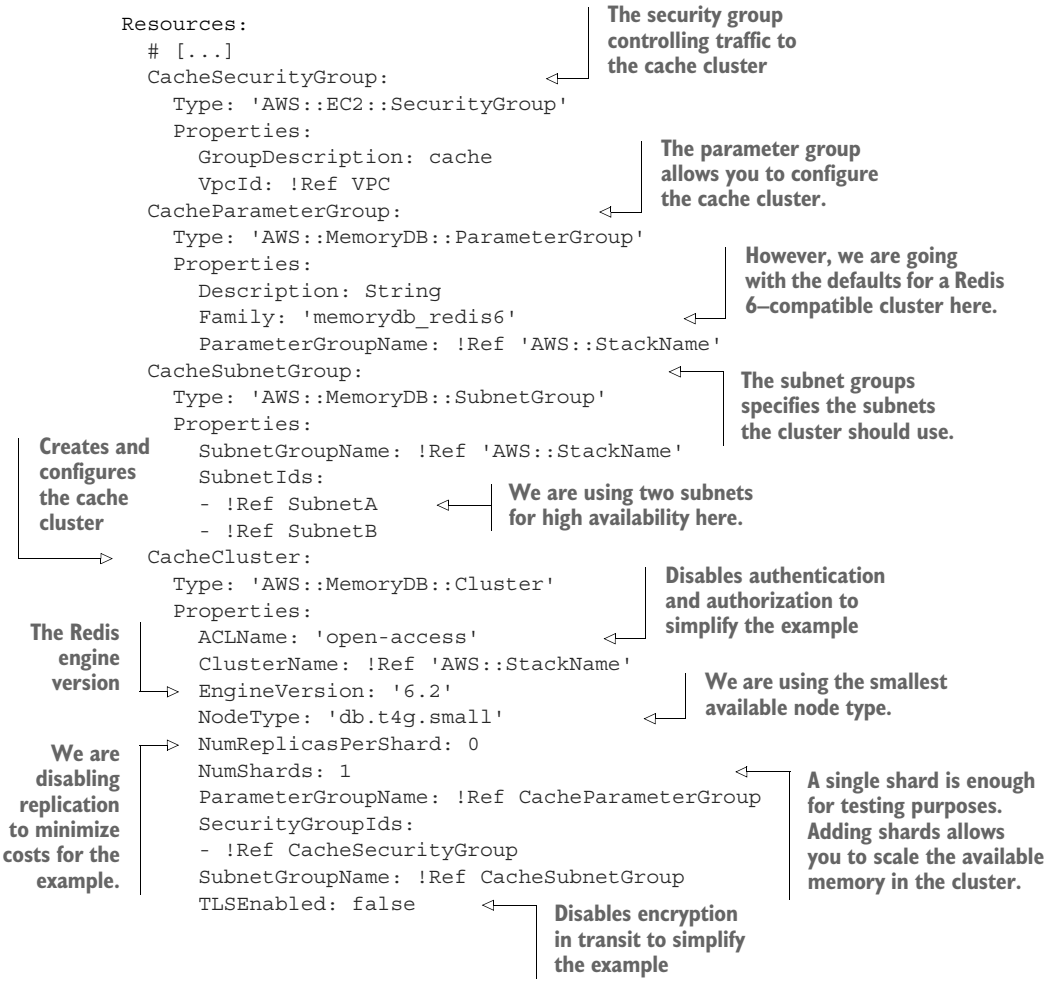
So, MemoryDB is a good fit when a key-value store is all you need, but data persistency is essential, such as the following situations:

- *Shopping cart*—Stores the items a user intends to check out
- *Content management system (CMS)*—Stores blog posts and comments
- *Device management service*—Stores status information about IoT devices

MemoryDB has been generally available since August 2021. We haven't used MemoryDB for production workloads yet, but the approach sounds very promising to us.

Want to give MemoryDB a try? We have prepared a simple example for you. Create a CloudFormation stack by using the Management Console: http://mng.bz/jAJy.

While you wait for the stack to reach state CREATE_COMPLETE, let's have a look into the CloudFormation template necessary to spin up a MemoryDB cluster, shown here:

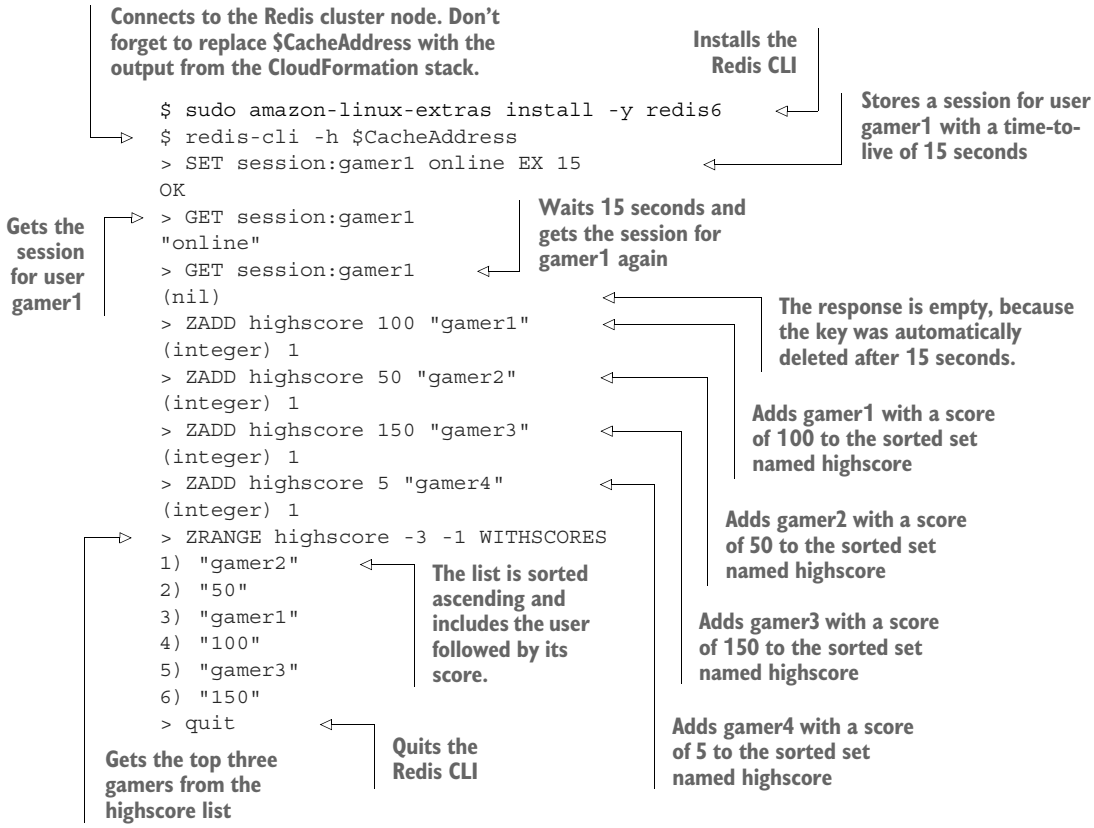


Want to have a deeper look into the CloudFormation template?

> **Where is the template located?**
>
> You can find the template on GitHub. Download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter11/memorydb-minimal.yaml. On S3, the same file is located at http://s3.amazonaws.com/awsinaction-code3/chapter11/memorydb-minimal.yaml.

We are reusing the gaming example from section 11.1. Hopefully, your CloudFormation stack memorydb-minimal reached status CREATE_COMPLETE already. If so, open the

Outputs tab and copy the EC2 instance ID as well as the cache address. Next, use the Session Manager to connect to the EC2 instance. Afterward, use the steps you used previously in section 11.1 to play with Redis. Make sure to replace `$CacheAddress` with the value from the CloudFormation outputs:



MemoryDB behaves like ElastiCache for Redis, but it comes with persistence guarantees, which make it possible to use it as the primary database.

Cleaning up

It's time to delete the CloudFormation stack named `memorydb-minimal`. Use the AWS Management Console or the following command to do so:

```
$ aws cloudformation delete-stack --stack-name memorydb-minimal.
```

You are now equipped to select the best-fitting in-memory database engine and deployment option for your use case. In the next section, you will take a closer look at the security aspects of ElastiCache to control access to your cache cluster.

11.3 Controlling cache access

Controlling access to data stored in ElastiCache is very similar to the way it works with RDS (see section 11.4). ElastiCache is protected by the following four layers:

- *Identity and Access Management (IAM)*—Controls which IAM user, group, or role is allowed to administer an ElastiCache cluster.
- *Security groups*—Restricts incoming and outgoing traffic to ElastiCache nodes.
- *Cache engine*—Redis >6.0 supports authentication and authorization with role-based access control (RBAC). Memcached does not support authentication.
- *Encryption*—Optionally, data can be encrypted at rest and in transit.

11.3.1 Controlling access to the configuration

Access to the ElastiCache service is controlled with the help of the IAM service. The IAM service is responsible for controlling access to actions like creating, updating, and deleting a cache cluster. IAM doesn't manage access inside the cache; that's the job of the cache engine. An IAM policy defines the configuration and management actions a user, group, or role is allowed to execute on the ElastiCache service. Attaching the IAM policy to IAM users, groups, or roles controls which entity can use the policy to configure an ElastiCache cluster. You can get a complete list of IAM actions and resource-level permissions supported at http://mng.bz/WM9x.

> **SECURITY WARNING** It's important to understand that you don't control access to the cache nodes using IAM. Once the nodes are created, security groups control the access on the network layer. Redis optionally supports user authentication and authorization.

11.3.2 Controlling network access

Network access is controlled with security groups. Remember the security group from the minimal CloudFormation template in section 11.1 where access to port 6379 (Redis) was allowed for all IP addresses. But because cluster nodes have only private IP addresses this restricts access to the VPC, as shown here:

```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        CidrIp: '0.0.0.0/0'
```

You should improve this setup by working with two security groups. To control traffic as tightly as possible, you will not allow traffic from certain IP addresses. Instead, you

create two security groups. The client security group will be attached to all EC2 instances communicating with the cache cluster (your web or application servers). The cache cluster security group allows inbound traffic on port 6379 only for traffic that comes from the client security group. This way you can have a dynamic fleet of clients who are allowed to send traffic to the cache cluster, as shown next. You used the same approach in section 5.4:

```
Resources:
  # [...]
  ClientSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'cache-client'
      VpcId: !Ref VPC
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        SourceSecurityGroupId: !Ref ClientSecurityGroup
```

Only allows access from the ClientSecurityGroup

Attach the `ClientSecurityGroup` to all EC2 instances that need access to the cache cluster. This way, you allow access only to the EC2 instances that really need access.

Keep in mind that ElastiCache nodes always have private IP addresses. This means that you can't accidentally expose a Redis or Memcached cluster to the internet. You still want to use security groups to implement the principle of least privilege.

11.3.3 Controlling cluster and data access

Unfortunately, ElastiCache for Memcached does not provide user authentication. However, you have two different ways to authenticate users with ElastiCache for Redis:

- Basic token-based authentication
- Users with RBAC

Use token-based authentication when all clients are allowed to read and manipulate all the data stored in the cluster. Use RBAC if you need to restrict access for different users, for example, to make sure the frontend is only allowed to read some of the data and the backend is able to write and read all the data. Keep in mind that when using authentication, you should also enable encryption in-transit to make sure to not transmit secrets in plain text.

In the next section, you'll learn how to use ElastiCache for Redis in a real-world application called Discourse.

11.4 Installing the sample application Discourse with CloudFormation

Small communities, like football clubs, reading circles, or dog schools, benefit from having a place where members can communicate with each other. Discourse is an open source software application for providing modern forums to your community. The forum software is written in Ruby using the Rails framework. Figure 11.8 gives you an impression of Discourse. Wouldn't that be a perfect place for your community to meet? In this section, you will learn how to set up Discourse with CloudFormation. Discourse is also perfectly suited for you to learn about ElastiCache because it requires a Redis in-memory database acting as a caching layer.

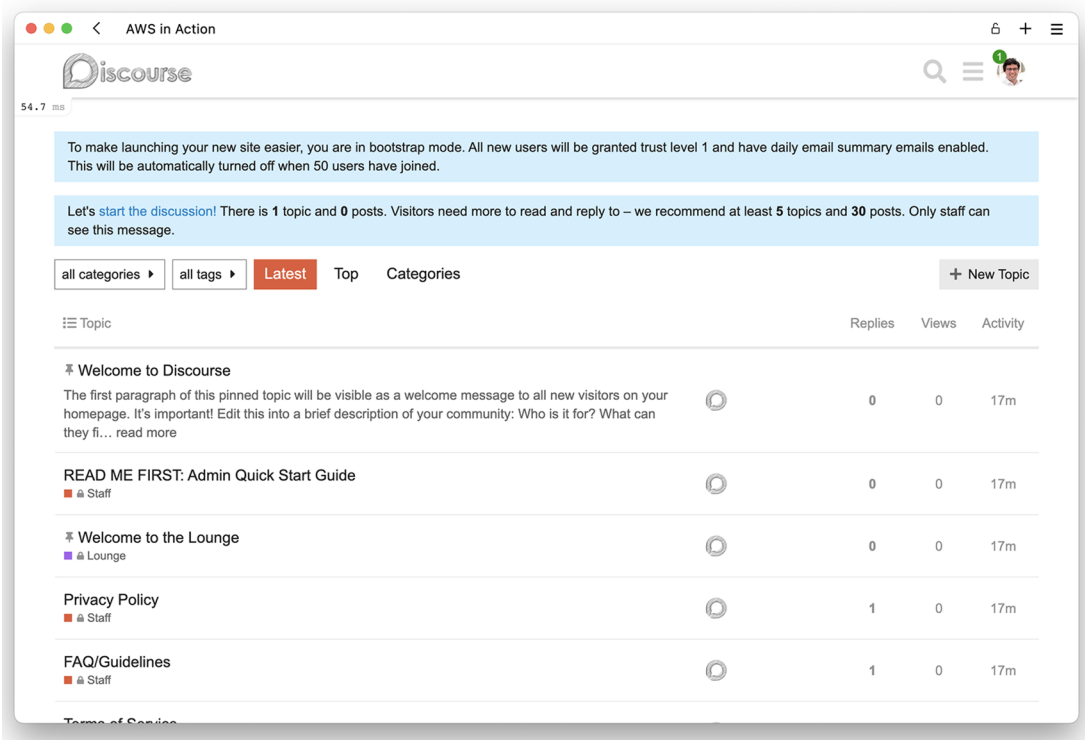


Figure 11.8 Discourse: A platform for community discussion

Discourse requires PostgreSQL as its main data store and uses Redis to cache data and process transient data. In this section, you'll create a CloudFormation template with all the components necessary to run Discourse. Finally, you'll create a CloudFormation stack based on the template to test your work. The necessary components follow:

- *VPC*—Network configuration
- *Cache*—Security group, subnet group, cache cluster

- *Database*—Security group, subnet group, database instance
- *Virtual machine*—Security group, EC2 instance

You'll start with the first component and extend the template in the rest of this section.

11.4.1 VPC: Network configuration

In section 5.5, you learned all about private networks on AWS. If you can't follow the next listing, you could go back to section 5.5 or continue with the next step—understanding the network is not key to get Discourse running.

Listing 11.2 CloudFormation template for Discourse: VPC

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 11'
Parameters:
  AdminEmailAddress:
    Description: 'Email address of admin user'
    Type: 'String'
Resources:
  VPC:
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: '172.31.0.0/16'
      EnableDnsHostnames: true
  InternetGateway:
    Type: 'AWS::EC2::InternetGateway'
    Properties: {}
  VPCGatewayAttachment:
    Type: 'AWS::EC2::VPCGatewayAttachment'
    Properties:
      VpcId: !Ref VPC
      InternetGatewayId: !Ref InternetGateway
  SubnetA:
    Type: 'AWS::EC2::Subnet'
    Properties:
      AvailabilityZone: !Select [0, !GetAZs '']
      CidrBlock: '172.31.38.0/24'
      VpcId: !Ref VPC
  SubnetB: # [...]
  RouteTable:
    Type: 'AWS::EC2::RouteTable'
    Properties:
      VpcId: !Ref VPC
  SubnetRouteTableAssociationA:
    Type: 'AWS::EC2::SubnetRouteTableAssociation'
    Properties:
      SubnetId: !Ref SubnetA
      RouteTableId: !Ref RouteTable
  # [...]
  RouteToInternet:
    Type: 'AWS::EC2::Route'
```

The email address of the Discourse admin must be valid.

Creates a VPC in the address range 172.31.0.0/16

We want to access Discourse from the internet, so we need an internet gateway.

Attaches the internet gateway to the VPC

Creates a subnet in the address range 172.31.38.0/24 in the first availability zone (array index 0)

Creates a second subnet in the address range 172.31.37.0/24 in the second availability zone (properties omitted)

Creates a route table that contains the default route, which routes all subnets in a VPC

Associates the first subnet with the route table

Adds a route to the internet via the internet gateway

```
    Properties:
      RouteTableId: !Ref RouteTable
      DestinationCidrBlock: '0.0.0.0/0'
      GatewayId: !Ref InternetGateway
    DependsOn: VPCGatewayAttachment
```

The following listing adds the required network access control list.

Listing 11.3 CloudFormation template for Discourse: VPC NACLs

```
Resources:
  # [...]
  NetworkAcl:                                    # Creates an empty network ACL
    Type: AWS::EC2::NetworkAcl
    Properties:
      VpcId: !Ref VPC
  SubnetNetworkAclAssociationA:                  # Associates the first subnet with the network ACL
    Type: 'AWS::EC2::SubnetNetworkAclAssociation'
    Properties:
      SubnetId: !Ref SubnetA
      NetworkAclId: !Ref NetworkAcl
  # [...]
  NetworkAclEntryIngress:                        # Allows all incoming traffic on the network ACL. (You will use security groups later as a firewall.)
    Type: 'AWS::EC2::NetworkAclEntry'
    Properties:
      NetworkAclId: !Ref NetworkAcl
      RuleNumber: 100
      Protocol: -1
      RuleAction: allow
      Egress: false
      CidrBlock: '0.0.0.0/0'
  NetworkAclEntryEgress:                         # Allows all outgoing traffic on the network ACL
    Type: 'AWS::EC2::NetworkAclEntry'
    Properties:
      NetworkAclId: !Ref NetworkAcl
      RuleNumber: 100
      Protocol: -1
      RuleAction: allow
      Egress: true
      CidrBlock: '0.0.0.0/0'
```

The network is now properly configured using two public subnets. Let's configure the cache next.

11.4.2 Cache: Security group, subnet group, cache cluster

You will add the ElastiCache for Redis cluster now. You learned how to describe a minimal cache cluster earlier in this chapter. This time, you'll add a few extra properties to enhance the setup. The next code listing contains the CloudFormation resources related to the cache.

Listing 11.4 CloudFormation template for Discourse: Cache

```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
  CacheSecurityGroupIngress:
    Type: 'AWS::EC2::SecurityGroupIngress'
    Properties:
      GroupId: !Ref CacheSecurityGroup
      IpProtocol: tcp
      FromPort: 6379
      ToPort: 6379
      SourceSecurityGroupId: !Ref InstanceSecurityGroup
  CacheSubnetGroup:
    Type: 'AWS::ElastiCache::SubnetGroup'
    Properties:
      Description: cache
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
  Cache:
    Type: 'AWS::ElastiCache::CacheCluster'
    Properties:
      CacheNodeType: 'cache.t2.micro'
      CacheSubnetGroupName: !Ref CacheSubnetGroup
      Engine: redis
      EngineVersion: '6.2'
      NumCacheNodes: 1
      VpcSecurityGroupIds:
      - !Ref CacheSecurityGroup
```

The security group to control incoming and outgoing traffic to/from the cache

To avoid a cyclic dependency, the ingress rule is split into a separate CloudFormation resource.

The InstanceSecurityGroup resource is not yet specified; you will add this later when you define the EC2 instance that runs the web server.

Redis runs on port 6379.

The cache subnet group references the VPC subnets.

Creates a single-node Redis cluster

You can specify the exact version of Redis that you want to run. Otherwise, the latest version is used, which may cause incompatibility issues in the future. We recommend always specifying the version.

The single-node Redis cache cluster is now defined. Discourse also requires a PostgreSQL database, which you'll define next.

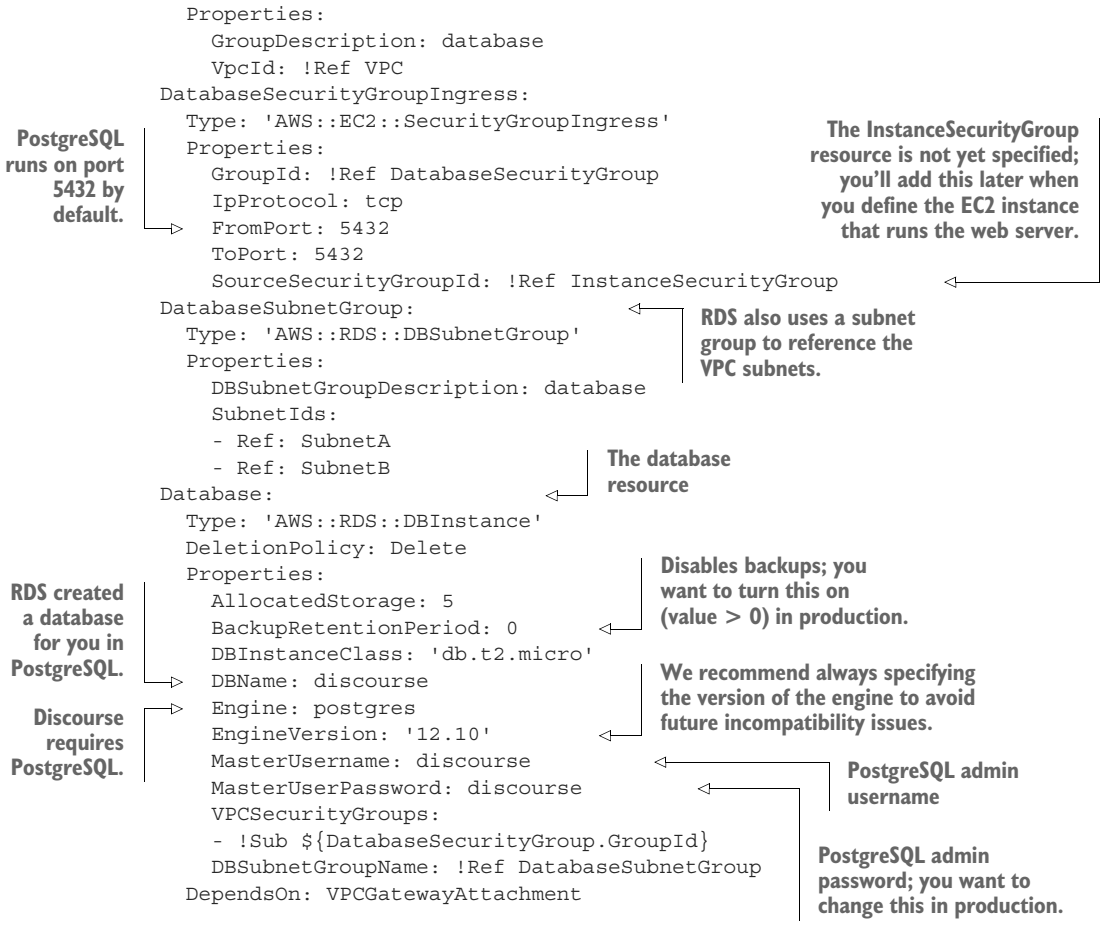
11.4.3 Database: Security group, subnet group, database instance

PostgreSQL is a powerful, open source relational database. If you are not familiar with PostgreSQL, that's not a problem at all. Luckily, the RDS service will provide a managed PostgreSQL database for you. You learned about RDS in chapter 10. The following listing shows the section of the template that defines the RDS instance.

Listing 11.5 CloudFormation template for Discourse: Database

```
Resources:
  # [...]
  DatabaseSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
```

Traffic to/from the RDS instance is protected by a security group.



Have you noticed the similarity between RDS and ElastiCache? The concepts are similar, which makes it easier for you to work with both services. Only one component is missing: the EC2 instance that runs the web server.

11.4.4 Virtual machine: Security group, EC2 instance

Discourse is a Ruby on Rails application, so you need an EC2 instance to host the application. Because it is very tricky to install the required Ruby environment with all its dependencies, we are using the recommended way to run Discourse, which is running a container with Docker on the virtual machine. However, running containers is not in scope of this chapter. You will learn more about deploying containers on AWS in chapter 18.

The next listing defines the virtual machine and the startup script to install and configure Discourse.

Discourse requires SMTP to send email

The forum software application Discourse requires access to an SMTP server to send email. Unfortunately, operating an SMTP server on your own is almost impossible, because most providers will not accept emails or flag them as spam. That's why we are using the Simple Email Service (SES) provided by AWS. SES requires verification before sending emails. Start with enabling sandbox mode by adding your email address to the allowlist as follows:

1. Open SES in the AWS Management Console.
2. Select Verified Identities from the menu.
3. Click Create Identity.
4. Choose the identity type Email Address.
5. Type in your email address.
6. Click Create Identity.
7. Check your inbox, and click the verification link.

Please note: you need to request production access to ensure SES is delivering emails to any address. To follow our example, however, this isn't needed.

Listing 11.6 CloudFormation template for Discourse: Virtual machine

```
Resources:
  # [...]
  InstanceSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'vm'
      SecurityGroupIngress:
      - CidrIp: '0.0.0.0/0'
        FromPort: 80
        IpProtocol: tcp
        ToPort: 80
      VpcId: !Ref VPC
  Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      InstanceType: 't2.micro'
      IamInstanceProfile: !Ref InstanceProfile
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref InstanceSecurityGroup
        SubnetId: !Ref SubnetA
      BlockDeviceMappings:
      - DeviceName: '/dev/xvda'
        Ebs:
          VolumeSize: 16
```

Allows HTTP traffic from the public internet

The virtual machine that runs Discourse

Increases the default volume size from 8 GB to 16 GB

```
      VolumeType: gp2
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -x
        bash -ex << "TRY"
          # [...]

          # install and start docker
          yum install -y git
          amazon-linux-extras install docker -y
          systemctl start docker

          docker run --restart always -d -p 80:80 --name discourse \
            -e "UNICORN_WORKERS=3" \
            # [...]
            -e "RUBY_ALLOCATOR=/usr/lib/libjemalloc.so.1" \
            public.ecr.aws/awsinaction/discourse:3rd /sbin/boot

          docker exec discourse /bin/sh -c /
            "cd /var/www/discourse && rake db:migrate"
          docker restart discourse
        TRY
        /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} /
          --resource Instance --region ${AWS::Region}
    CreationPolicy:
      ResourceSignal:
        Timeout: PT15M
    DependsOn:
    - VPCGatewayAttachment
```

Installs and starts Docker

Creates and launches a Docker container

Runs the database migration script to initialize or migrate the database

Restarts the container to make sure the database migration is in effect

You've reached the end of the template. All components are defined now. It's time to create a CloudFormation stack based on your template to see whether it works.

11.4.5 Testing the CloudFormation template for Discourse

Let's create a stack based on your template to create all the resources in your AWS account, as shown next. To find the full code for the template, go to /chapter11/discourse.yaml in the book's code folder. Use the AWS CLI to create the stack. Don't forget to replace `$AdminEmailAddress` with your e-mail address:

```
$ aws cloudformation create-stack --stack-name discourse \
➥ --template-url https://s3.amazonaws.com/\
➥awsinaction-code3/chapter11/discourse.yaml \
➥ --parameters \
➥ "ParameterKey=AdminEmailAddress,ParameterValue=$AdminEmailAddress" \
➥ --capabilities CAPABILITY_NAMED_IAM
```

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter11/discourse.yaml. On S3, the same file is located at https://s3.amazonaws.com/awsinaction-code3/chapter11/discourse.yaml.

The creation of the stack can take up to 20 minutes. After the stack has been created, get the public IP address of the EC2 instance from the stack's output by executing the following command:

```
$ aws cloudformation describe-stacks --stack-name discourse \
➥ --query "Stacks[0].Outputs[1].OutputValue"
```

Open a web browser and insert the IP address in the address bar to open your Discourse website. Figure 11.9 shows the website. Click Register to create an admin account.

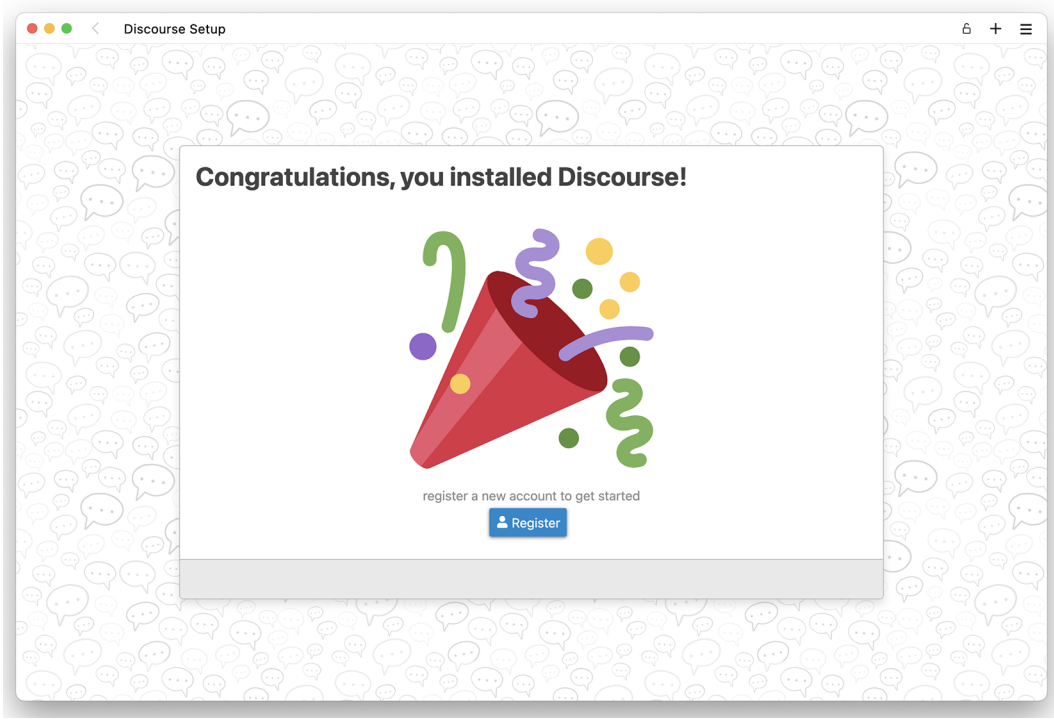


Figure 11.9 Discourse: First screen after a fresh install

You will receive an email to activate your account. Note that this email might be hidden in your spam folder. After activation, the setup wizard is started, which you have to complete to configure your forum. After you complete the wizard and have successfully installed Discourse, don't delete the CloudFormation stack because you'll use the setup in the next section.

Congratulations—you have deployed Discourse, a rather complex web application, using Redis as a cache to speed up loading times. You've experienced that ElastiCache is for in-memory databases, like RDS is for relational databases. Because ElastiCache is a relevant part of the architecture, you need to make sure you monitor the cache

infrastructure to avoid performance issues or even downtimes. You will learn about monitoring ElastiCache in the following section.

11.5 Monitoring a cache

CloudWatch is the service on AWS that stores all kinds of metrics. Like other services, ElastiCache nodes send metrics to CloudWatch as well. The most important metrics to watch follow:

- `CPUUtilization`—The percentage of CPU utilization.
- `SwapUsage`—The amount of swap used on the host, in bytes. *Swap* is space on disk that is used if the system runs out of physical memory.
- `Evictions`—The number of nonexpired items the cache evicted due to the memory limit.
- `ReplicationLag`—This metric is applicable only for a Redis node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. Usually this number is very low.

In this section we'll examine those metrics in more detail and give you some hints about useful thresholds for defining alarms on those metrics to set up production-ready monitoring for your cache.

11.5.1 Monitoring host-level metrics

The virtual machines running underneath the ElastiCache service report CPU utilization and swap usage. CPU utilization usually gets problematic when crossing 80–90%, because the wait time explodes. But things are more tricky here. Redis is single-threaded. If you have many cores, the overall CPU utilization can be low, but one core can be at 100% utilization. Use the `EngineCPUUtilization` metric to monitor the Redis process CPU utilization. Swap usage is a different topic. You run an in-memory cache, so if the virtual machine starts to swap (move memory to disk), the performance will suffer. By default, ElastiCache for Memcached and Redis is configured to limit memory consumption to a value smaller than what's physical available (you can tune this) to have room for other resources (e.g., the kernel needs memory for each open socket). But other processes (such as kernel processes) are also running, and they may start to consume more memory than what's available. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards.

Queuing theory: Why 80–90%?

Imagine you are the manager of a supermarket. What should be the goal for daily utilization of your cashiers? It's tempting to go for a high number, maybe 90%. But it turns out that the wait time for your customers is very high when your cashiers are used for 90% of the day, because customers don't arrive at the same time at the queue. The theory behind this is called *queuing theory*, and it turns out that wait time is exponential

(continued)

to the utilization of a resource. This applies not only to cashiers but also to network cards, CPU, hard disks, and so on. Keep in mind that this sidebar simplifies the theory and assumes an M/D/1 queuing system: Markovian arrivals (exponentially distributed arrival times), deterministic service times (fixed), one service center. To learn more about queuing theory applied to computer systems, we recommend *Systems Performance: Enterprise and the Cloud* by Brendan Gregg (Prentice Hall, 2013) to get started.

When you go from 0% utilization to 60%, wait time doubles. When you go to 80%, wait time has tripled. When you to 90%, wait time is six times higher, and so on. If your wait time is 100 ms during 0% utilization, you already have 300 ms wait time during 80% utilization, which is already slow for an e-commerce website.

You might set up an alarm to trigger if the 10-minute average of the `EngineCPUUtilization` metric is higher than 80% for one out of one data points, and if the 10-minute average of the `SwapUsage` metric is higher than 67108864 (64 MB) for one out of one data points. These numbers are just a rule of thumb. You should load-test your system to verify that the thresholds are high or low enough to trigger the alarm before application performance suffers.

11.5.2 Is my memory sufficient?

The `Evictions` metric is reported by Memcached and Redis. If the cache is running out of memory and you try to add a new key-value pair, an old key-value pair needs to be deleted first. This is called an eviction. By default, Redis deletes the least recently used key, but only for keys that define a TTL. This strategy is called `volatile-lru`. On top of that, the following eviction strategies are available:

- `allkeys-lru`—Removes the least recently used key among all keys
- `volatile-random`—Removes a random key among keys with TTL
- `allkeys-random`—Removes a random key among all keys
- `volatile-ttl`—Removes the key with the shortest TTL
- `noeviction`—Does not evict any key

Usually, high eviction rates are a sign that you either aren't using a TTL to expire keys after some time or that your cache is too small. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards. You might set an alarm to trigger if the 10-minute average of the `Evictions` metric is higher than 1000 for one out of one data points.

11.5.3 Is my Redis replication up-to-date?

The `ReplicationLag` metric is applicable only for a node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. The higher this value, the more out of date the replica is. This can be a

problem because some users of your application will see very old data. In the gaming application, imagine you have one primary node and one replica node. All reads are performed by either the primary or the replica node. The `ReplicationLag` is 600, which means that the replication node looks like the primary node looked 10 minutes before. Depending on which node the user hits when accessing the application, they could see 10-minute-old data.

What are reasons for a high `ReplicationLag`? There could be a problem with the sizing of your cluster—for example, your cache cluster might be at capacity. Typically this will be a sign to increase the capacity by adding shards or replicas. You might set an alarm to trigger if the 10-minute average of the `ReplicationLag` metric is higher than 30 for one consecutive period.

Cleaning up

It's time to delete the running CloudFormation stack, as shown here:

```
$ aws cloudformation delete-stack --stack-name discourse
```

11.6 Tweaking cache performance

Your cache can become a bottleneck if it can no longer handle the requests with low latency. In the previous section, you learned how to monitor your cache. In this section, you learn what you can do if your monitoring data shows that your cache is becoming the bottleneck (e.g., if you see high CPU or network usage). Figure 11.10 contains a decision tree that you can use to resolve performance issues with ElastiCache. The strategies are described in more detail in the rest of this section.

Three strategies for tweaking the performance of your ElastiCache cluster follow:

- *Selecting the right cache node type*—A bigger instance type comes with more resources (CPU, memory, network) so you can scale vertically.
- *Selecting the right deployment option*—You can use sharding or read replicas to scale horizontally.
- *Compressing your data*—If you shrink the amount of data being transferred and stored, you can also tweak performance.

11.6.1 Selecting the right cache node type

So far, you used the cache node type `cache.t2.micro`, which comes with one vCPU, ~0.6 GB memory, and low to moderate network performance. You used this node type because it's part of the Free Tier. But you can also use more powerful node types on AWS. The upper end is the `cache.r6gd.16xlarge` with 64 vCPUs, ~419 GB memory, and 25 Gb network.

As a rule of thumb: for production traffic, select a cache node type with at least 2 vCPUs for real concurrency, enough memory to hold your data set with some space to

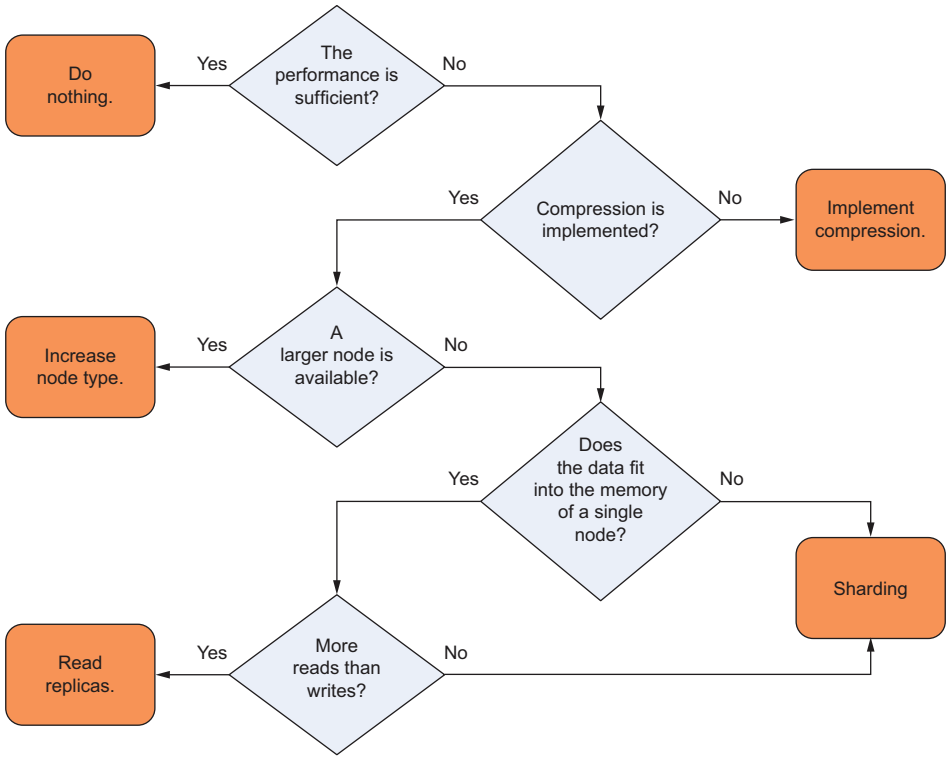


Figure 11.10 An ElastiCache decision tree to resolve performance issues

grow (say, 20%; this also avoids memory fragmentation), and at least high network performance. The `cache.r6g.large` is an excellent choice for a small node size: 2 vCPUs, ~13 GB, and up to 10 GB of network. This may be a good starting point when considering how many shards you may want in a clustered topology, and if you need more memory, move up a node type. You can find the available node types at https://aws.amazon.com/elasticache/pricing/.

11.6.2 Selecting the right deployment option

By replicating data, you can distribute read traffic to multiple nodes within the same replica group. By replicating data from the primary node to one or multiple replication nodes, you are increasing the number of nodes that are able to process read requests and, therefore, increase the maximum throughput. On top of that, by sharding data, you split the data into multiple buckets. Each bucket contains a subset of the data. Each shard is stored on a primary node and is optionally synchronized to replication nodes as well. This further increases the number of nodes available to answer incoming reads—and even writes.

Redis supports the concept of replication, where one node in a node group is the primary node accepting read and write traffic, while the replica nodes only accept read traffic, which allows you to scale the read capacity. The Redis client has to be aware of the cluster topology to select the right node for a given command. Keep in mind that the replicas are synchronized asynchronously. This means that the replication node eventually reaches the state of the primary node.

Both Memcached and Redis support the concept of sharding. With sharding, a single cache cluster node is no longer responsible for all the keys. Instead the key space is divided across multiple nodes. Both Redis and Memcached clients implement a hashing algorithm to select the right node for a given key. By sharding, you can increase the capacity of your cache cluster.

As a rule of thumb: when a single node can no longer handle the amount of data or the requests, and if you are using Redis with mostly read traffic, then you should use replication. Replication also increases the availability at the same time (at no extra cost).

It is even possible to increase and decrease the number of shards automatically, based on utilization metrics, by using Application Auto Scaling. See http://mng.bz/82d2 to learn more.

Last but not least, replicating data is possible not only within a cluster but also between clusters running in other regions of the world. For example, if you are operating a popular online game, it is necessary to minimize latency by deploying your cloud infrastructure to the United States, Europe, and Asia. By using ElastiCache Global Datastore, you are able to replicate a Redis cluster located in Ireland to North Virginia and Singapore, for example.

11.6.3 Compressing your data

Instead of sending large values (and also keys) to your cache, you can compress the data before you store it in the cache. When you retrieve data from the cache, you have to uncompress it on the application before you can use the data. Depending on your data, compressing data can have a significant effect. We saw memory reductions to 25% of the original size and network transfer savings of the same size. Please note that this approach needs to be implemented in your application.

As a rule of thumb: compress your data using an algorithm that is best suited for your data. Most likely the `zlib` library is a good starting point. You might want to experiment with a subset of your data to select the best compression algorithm that is also supported by your programming language.

On top of that, ElastiCache for Redis also supports data tiering to reduce costs. With data tiering enabled, ElastiCache will move parts of your data from memory to solid-state disks—a tradeoff between costs and latency. Using data tiering is most interesting if your workload reads only parts of the data frequently.

Summary

- A caching layer can speed up your application significantly, while also lowering the costs of your primary data store.
- To keep the cache in sync with the database, items usually expire after some time, or a write-through strategy is used.
- When the cache is full, the least frequently used items are usually evicted.
- ElastiCache can run Memcached or Redis clusters for you. Depending on the engine, different features are available. Memcached and Redis are open source, but AWS added engine-level enhancements.
- Memcached is a simple key-value store allowing you to scale available memory and throughput by adding additional machines—called shards—to the cluster.
- Redis is a more advanced in-memory database that supports complex data structures like sorted sets.
- MemoryDB is an alternative to ElastiCache and provides an in-memory database with strong persistence guarantees. MemoryDB comes with Redis compatibility.
- CloudWatch provides insights into the utilization and performance of ElastiCache.

12 Programming for the NoSQL database service: DynamoDB

This chapter covers

- Advantages and disadvantages of the NoSQL service, DynamoDB
- Creating tables and storing data
- Adding secondary indexes to optimize data retrieval
- Designing a data model optimized for a key-value database
- Optimizing costs by choosing the best fitting capacity mode and storage type

Most applications depend on a database to store data. Imagine an application that keeps track of a warehouse's inventory. The more inventory moves through the warehouse, the more requests the application serves and the more queries the database has to process. Sooner or later, the database becomes too busy and latency increases to a level that limits the warehouse's productivity. At this point, you have to scale the database to help the business. You can do this in two ways:

- *Vertically*—You can use faster hardware for your database machine; for example, you can add memory or replace the CPU with a more powerful model.
- *Horizontally*—You can add a second database machine. Both machines then form a *database cluster*.

Scaling a database vertically is the easier option, but it gets expensive. High-end hardware is more expensive than commodity hardware. Besides that, at some point, you will not find more powerful machines on the market anymore.

Scaling a traditional relational database horizontally is difficult because transactional guarantees such as atomicity, consistency, isolation, and durability—also known as *ACID*—require communication among all nodes of the database during a two-phase commit. To see what we're talking about, here's how a simplified two-phase commit with two nodes works, as illustrated in figure 12.1:

1. A query is sent to the database cluster that wants to change data (`INSERT`, `UPDATE`, `DELETE`).
2. The database transaction coordinator sends a commit request to the two nodes.
3. Node 1 checks whether the query could be executed. The decision is sent back to the coordinator. If the nodes decide yes, the query can be executed, it must fulfill this promise. There is no way back.
4. Node 2 checks whether the query could be executed. The decision is sent back to the coordinator.
5. The coordinator receives all decisions. If all nodes decide that the query could be executed, the coordinator instructs the nodes to finally commit.
6. Nodes 1 and 2 finally change the data. At this point, the nodes must fulfill the request. This step must not fail.

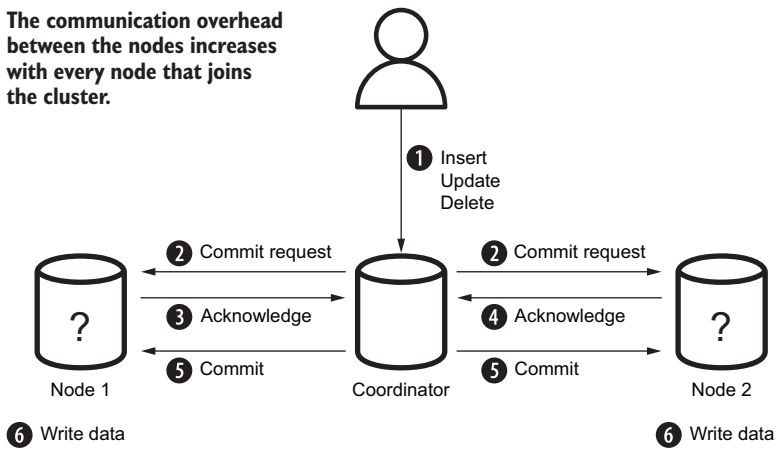


Figure 12.1 The communication overhead between the nodes increases with every node that joins the cluster.

The problem is, the more nodes you add, the slower your database becomes, because more nodes must coordinate transactions between each other. The way to tackle this has been to use databases that don't adhere to these guarantees. They're called *NoSQL databases*.

There are four types of NoSQL databases—document, graph, columnar, and key-value store—each with its own uses and applications. Amazon provides a NoSQL database service called *DynamoDB*, a key-value store. DynamoDB is a fully managed, proprietary, closed source key-value store with document support. In other words, DynamoDB persists objects identified by a unique key, which you might know from the concept of a hash table. *Fully managed* means that you only use the service and AWS operates it for you. DynamoDB is highly available and highly durable. You can scale from one item to billions and from one request per second to tens of thousands of requests per second. AWS also offers other types of NoSQL database systems like Keyspaces, Neptune, DocumentDB, and MemoryDB for Redis—more about these options later.

To use DynamoDB, your application needs to be built for this particular NoSQL database. You cannot point your legacy application to DynamoDB instead of a MySQL database, for example. Therefore, this chapter focuses on how to write an application for storing and retrieving data from DynamoDB. At the end of the chapter, we will discuss what is needed to administer DynamoDB.

Some typical use cases for DynamoDB follow:

- When building systems that need to deal with a massive amount of requests or spiky workloads, the ability to scale horizontally is a game changer. We have used DynamoDB to track client-side errors from a web application, for example.
- When building small applications with a simple data structure, the pay-per-request pricing model and the simplicity of a fully managed service are good reasons to go with DynamoDB. For example, we used DynamoDB to track the progress of batch jobs.

While following this chapter, you will implement a simple to-do application called nodetodo, the equivalent of a "Hello World" example for databases. You will learn how to write, fetch, and query data. Figure 12.2 shows nodetodo in action.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. Keep in mind that this applies only if there is nothing else going on in your AWS account. You'll clean up your account at the end of the chapter.

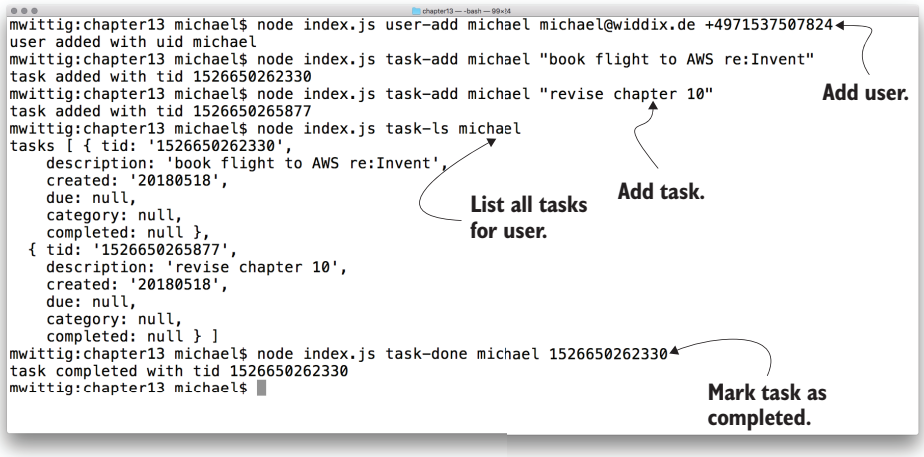


Figure 12.2 Manage your tasks with the command-line to-do application, nodetodo.

12.1 Programming a to-do application

DynamoDB is a key-value store that organizes your data into tables. For example, you can have a table to store your users and another table to store tasks. The items contained in the table are identified by a unique key. An item could be a user or a task; think of an item as a row in a relational database.

To minimize the overhead of a programming language, you'll use Node.js/JavaScript to create a small to-do application, nodetodo, which you can use via the terminal on your local machine. nodetodo uses DynamoDB as a database and comes with the following features:

- Creates and deletes users
- Creates and deletes tasks
- Marks tasks as done
- Gets a list of all tasks with various filters

To implement an intuitive CLI, nodetodo uses *docopt*, a command-line interface description language, to describe the CLI interface. The supported commands follow:

- `user-add`—Adds a new user to nodetodo
- `user-rm`—Removes a user
- `user-ls`—Lists users
- `user`—Shows the details of a single user
- `task-add`—Adds a new task to nodetodo
- `task-rm`—Removes a task
- `task-ls`—Lists user tasks with various filters

- `task-la`—Lists tasks by category with various filters
- `task-done`—Marks a task as finished

In the following sections, you'll implement these commands to learn about DynamoDB hands-on. This listing shows the full CLI description of all the commands, including parameters.

Listing 12.1 CLI description language docopt: Using nodetodo (cli.txt)

```
nodetodo

Usage:
  nodetodo user-add <uid> <email> <phone>
  nodetodo user-rm <uid>
  nodetodo user-ls [--limit=<limit>] [--next=<id>]
  nodetodo user <uid>
  nodetodo task-add <uid> <description> \
➥ [<category>] [--dueat=<yyyymmdd>]
  nodetodo task-rm <uid> <tid>
  nodetodo task-ls <uid> [<category>] \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-la <category> \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-done <uid> <tid>
  nodetodo -h | --help
  nodetodo --version

Options:
  -h --help         Show this screen.
  --version         Show version.
```

The named parameters limit and next are optional.

The category parameter is optional.

Pipe indicates either/or.

Version information

help prints information about how to use nodetodo.

DynamoDB isn't comparable to a traditional relational database in which you create, read, update, or delete data with SQL. You'll use the AWS SDK to send requests to the REST API. You must integrate DynamoDB into your application; you can't take an existing application that uses an SQL database and run it on DynamoDB. To use DynamoDB, you need to write code!

12.2 Creating tables

Each DynamoDB table has a name and organizes a collection of items. An *item* is a collection of attributes, and an *attribute* is a name-value pair. The attribute value can be scalar (number, string, binary, Boolean), multivalued (number set, string set, binary set), or a JSON document (object, array). Items in a table aren't required to have the same attributes; there is no enforced schema. Figure 12.3 demonstrates these terms.

DynamoDB doesn't need a static schema like a relational database does, but you must define the attributes that are used as the primary key in your table. In other words, you must define the table's primary key schema. Next, you will create a table for the users of the nodetodo application as well as a table that will store all the tasks.

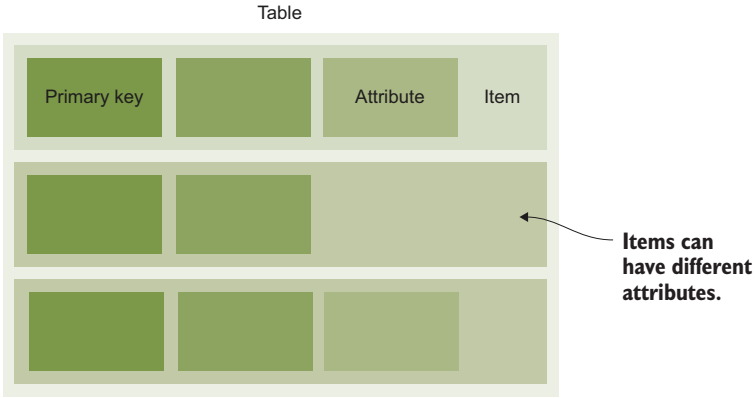


Figure 12.3 DynamoDB tables store items consisting of attributes identified by a primary key.

12.2.1 Users are identified by a partition key

To be able to assign a task to a user, the nodetodo application needs to store some information about users. Therefore, we came up with the following data structure for storing information about users:

```
{
  "uid": "emma",
  "email": "emma@widdix.de",
  "phone": "0123456789"
}
```

A unique user ID

The user's email address

The phone number belonging to the user

How to create a DynamoDB table based on this information? First, you need to think about the table's name. We suggest that you prefix all your tables with the name of your application. In this case, the table name would be `todo-user`.

Next, a DynamoDB table requires the definition of a *primary key*. A primary key consists of one or two attributes. A primary key is unique within a table and identifies an item. You need the primary key to retrieve, update, or delete an item. Note that DynamoDB does not care about any other attributes of an item because it does not require a fixed schema, as relational databases do.

For the `todo-user` table, we recommend you use the `uid` as the primary key because the attribute is unique. Also, the only command that queries data is `user`, which fetches a user by its `uid`. When using a single attribute as primary key, DynamoDB calls this the *partition key* of the table.

You can create a table by using the Management Console, CloudFormation, SDKs, or the CLI. Within this chapter, we use the AWS CLI. The `aws dynamodb create-table` command has the following four mandatory options:

- `table-name`—Name of the table (can't be changed).
- `attribute-definitions`—Name and type of attributes used as the primary key. Multiple definitions can be given using the syntax `AttributeName=attr1, AttributeType=S`, separated by a space character. Valid types are `S` (string), `N` (number), and `B` (binary).
- `key-schema`—Name of attributes that are part of the primary key (can't be changed). Contains a single entry using the syntax `AttributeName=attr1, KeyType=HASH` for a partition key, or two entries separated by spaces for a partition key and sort key. Valid types are `HASH` and `RANGE`.
- `provisioned-throughput`—Performance settings for this table defined as `ReadCapacityUnits=5,WriteCapacityUnits=5` (you'll learn about this in section 12.11).

Execute the following command to create the `todo-user` table with the `uid` attribute as the partition key:

Prefixing tables with the name of your application will prevent name clashes in the future.

Items must at least have one attribute uid of type string.

```
$ aws dynamodb create-table --table-name todo-user \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ --key-schema AttributeName=uid,KeyType=HASH \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

The partition key (type HASH) uses the uid attribute.

You'll learn about this in section 12.11.

Creating a table takes some time. Wait until the status changes to `ACTIVE`. You can check the status of a table as follows.

Listing 12.2 Checking the status of the DynamoDB table

```
$ aws dynamodb describe-table --table-name todo-user
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "uid",
        "AttributeType": "S"
      }
    ],
    "TableName": "todo-user",
    "KeySchema": [
      {
        "AttributeName": "uid",
        "KeyType": "HASH"
      }
    ],
    "TableStatus": "ACTIVE",
    "CreationDateTime": "2022-01-24T16:00:29.105000+01:00",
```

The CLI command to check the table status

Attributes defined for that table

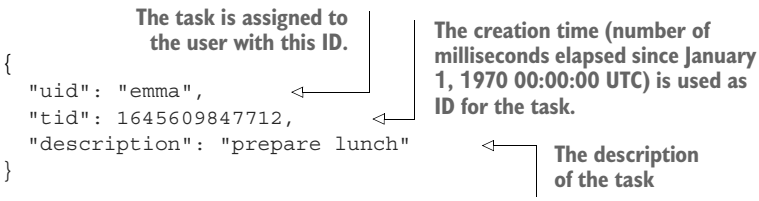
Attributes used as the primary key

Status of the table

```
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 5,
      "WriteCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:us-east-1:111111111111:table/todo-user",
    "TableId": "0697ea25-5901-421c-af29-8288a024392a"
  }
}
```

12.2.2 Tasks are identified by a partition key and sort key

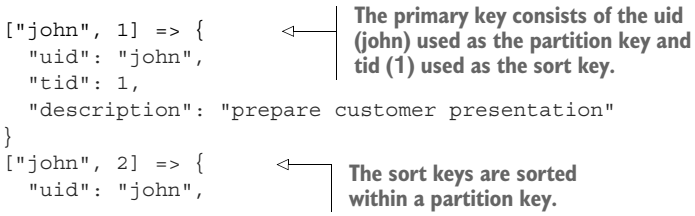
So far, we created the table `todo-user` to store information about the users of nodetodo. Next, we need a table to store the tasks. A task belongs to a user and contains a description of the task. Therefore, we came up with the following data structure:



How does nodetodo query the task table? By using the `task-ls` command, which we need to implement. This command lists all the tasks belonging to a user. Therefore, choosing the `tid` as the primary key is not sufficient. We recommend a combination of `uid` and `tid` instead. DynamoDB calls the components of a primary key with two attributes partition key and *sort key*. Neither the partition key nor the sort key need to be unique, but the combination of both parts must be unique.

> **NOTE** This solution has one limitation: users can add only one task per timestamp. Because tasks are uniquely identified by `uid` and `tid` (the primary key) there can't be two tasks for the same user at the same time. Our timestamp comes with millisecond resolution, so it should be fine.

Using a partition key and a sort key uses two of your table's attributes. For the partition key, an unordered hash index is maintained; the sort key is kept in a sorted index for each partition key. The combination of the partition key and the sort key uniquely identifies an item if they are used as the primary key. The following data set shows the combination of unsorted partition keys and sorted sort keys:



```
  "tid": 2,
  "description": "plan holidays"
}
["emma", 1] => {
  "uid": "emma",
  "tid": 1,
  "description": "prepare lunch"
}
["emma", 2] => {
  "uid": "emma",
  "tid": 2,
  "description": "buy nice flowers for mum"
}
["emma", 3] => {
  "uid": "emma",
  "tid": 3,
  "description": "prepare talk for conference"
}
```

There is no order in the partition keys.

Execute the following command to create the `todo-task` table with a primary key consisting of the partition key `uid` and sort key `tid` like this:

At least two attributes are needed for a partition key and sort key.

```
$ aws dynamodb create-table --table-name todo-task \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ --key-schema AttributeName=uid,KeyType=HASH \
➥ AttributeName=tid,KeyType=RANGE \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

The tid attribute is the sort key.

Wait until the table status changes to `ACTIVE` when you run `aws dynamodb describe-table --table-name todo-task`.

The `todo-user` and `todo-task` tables are ready. Time to add some data.

12.3 Adding data

Now you have two tables up and running to store users and their tasks. To use them, you need to add data. You'll access DynamoDB via the Node.js SDK, so it's time to set up the SDK and some boilerplate code before you implement adding users and tasks.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify that everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples like nodetodo for AWS.

(continued)

Do you want to learn more about Node.js? We recommend *Node.js in Action* (second edition) by Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* by PJ Evans (Manning, 2018).

To get started with Node.js and docopt, you need some magic lines to load all the dependencies and do some configuration work. Listing 12.3 shows how this can be done.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code3. nodetodo is located in /chapter12/. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

Docopt is responsible for reading all the arguments passed to the process. It returns a JavaScript object, where the arguments are mapped to the parameters in the CLI description.

Listing 12.3 nodetodo: Using docopt in Node.js (index.js)

```
const fs = require('fs');
const docopt = require('docopt');
const moment = require('moment');
const AWS = require('aws-sdk');
const db = new AWS.DynamoDB({
  region: 'us-east-1'
});

const cli = fs.readFileSync('./cli.txt',
➥ {encoding: 'utf8'});
const input = docopt.docopt(cli, {
  version: '1.0',
  argv: process.argv.splice(2)
});
```

Loads the fs module to access the filesystem

Loads the docopt module to read input arguments

Loads the moment module to simplify temporal types in JavaScript

Loads the AWS SDK module

Reads the CLI description from the file cli.txt

Parses the arguments, and saves them to an input variable

Next, you will add an item to a table. The next listing explains the `putItem` method of the AWS SDK for Node.js.

Listing 12.4 DynamoDB: Creating an item

```
const params = {
  Item: {
    attr1: {S: 'val1'},
    attr2: {N: '2'}
```

All item attribute name-value pairs

Strings are indicated by an S.

Numbers (floats and integers) are indicated by an N.

```
  },
  TableName: 'app-entity'
};
db.putItem(params, (err) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('success');
  }
});
```

Adds item to the app-entity table

Handles errors

Invokes the putItem operation on DynamoDB

The first step is to add an item to the `todo-user` table.

12.3.1 Adding a user

The following listing shows the code executed by the `user-add` command.

Listing 12.5 nodetodo: Adding a user (index.js)

```
if (input['user-add'] === true) {
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      email: {S: input['<email>']},
      phone: {S: input['<phone>']}
    },
    TableName: 'todo-user',
    ConditionExpression: 'attribute_not_exists(uid)'
  };
  db.putItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('user added');
    }
  });
}
```

Item contains all attributes. Keys are also attributes, and that's why you do not need to tell DynamoDB which attributes are keys if you add data.

The uid attribute is of type string and contains the uid parameter value.

The email attribute is of type string and contains the email parameter value.

The phone attribute is of type string and contains the phone parameter value.

Specifies the user table

If putItem is called twice on the same key, data is replaced. ConditionExpression allows the putItem only if the key isn't yet present.

Invokes the putItem operation on DynamoDB

When you make a call to the AWS API, you always do the following:

1. Create a JavaScript object (map) filled with the needed parameters (the params variable).
2. Invoke the function via the AWS SDK.
3. Check whether the response contains an error, and if not, process the returned data.

Therefore, you only need to change the content of `params` if you want to add a task instead of a user. Execute the following commands to add two users:

```
node index.js user-add john john@widdix.de +11111111
node index.js user-add emma emma@widdix.de +22222222
```

Time to add tasks.

12.3.2 Adding a task

John and Emma want to add tasks to their to-do lists to better organize their everyday life. Adding a task is similar to adding a user. The next listing shows the code used to create a task.

Listing 12.6 nodetodo: Adding a task (index.js)

```
if (input['task-add'] === true) {
  const tid = Date.now();
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      tid: {N: tid.toString()},
      description: {S: input['<description>']},
      created: {N: moment(tid).format('YYYYMMDD')}
    },
    TableName: 'todo-task',
    ConditionExpression: 'attribute_not_exists(uid)
    ➥ and attribute_not_exists(tid)'
  };
  if (input['--dueat'] !== null) {
    params.Item.due = {N: input['--dueat']};
  }
  if (input['<category>'] !== null) {
    params.Item.category = {S: input['<category>']};
  }
  db.putItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task added with tid ' + tid);
    }
  });
}
```

Creates the task ID (tid) based on the current timestamp

The tid attribute is of type number and contains the tid value.

The created attribute is of type number (format 20150525).

Specifies the task table

Ensures that an existing item is not overridden

If the optional named parameter dueat is set, adds this value to the item

If the optional named parameter category is set, adds this value to the item

Invokes the putItem operation on DynamoDB

Now we'll add some tasks. Use the following command to remind Emma about buying some milk and a task asking John to put out the garbage:

```
node index.js task-add emma "buy milk" "shopping"
node index.js task-add emma "put out the garbage" "housekeeping" --dueat "20220224"
```

Now that you are able to add users and tasks to nodetodo, wouldn't it be nice if you could retrieve all this data?

12.4 Retrieving data

So far, you have learned how to insert users and tasks into two different DynamoDB tables. Next, you will learn how to query the data, for example, to get a list of all tasks assigned to Emma.

DynamoDB is a key-value store. The key is usually the only way to retrieve data from such a store. When designing a data model for DynamoDB, you must be aware of that limitation when you create tables (as you did in section 12.2). If you can use only the key to look up data, you'll sooner or later experience difficulties. Luckily, DynamoDB provides two other ways to look up items: a secondary index lookup and the scan operation. You'll start by retrieving data by the items' primary key and continue with more sophisticated methods of data retrieval.

DynamoDB Streams

DynamoDB lets you retrieve changes to a table as soon as they're made. A Stream provides all write (create, update, delete) operations to your table items. The order is consistent within a partition key. DynamoDB Streams also help with the following:

- If your application polls the database for changes, DynamoDB Streams solves the problem in a more elegant way.
- If you want to populate a cache with the changes made to a table, DynamoDB Streams can help.

12.4.1 Getting an item by key

Let's start with a simple example. You want to find out the contact details about Emma, which are stored in the `todo-user` table.

The simplest form of data retrieval is looking up a single item by its primary key, for example, a user by its ID. The `getItem` SDK operation to get a single item from DynamoDB can be used like this.

Listing 12.7 DynamoDB: Query a single item (index.js)

```
const params = {
  Key: {
    attr1: {S: 'val1'}
  },
  TableName: 'app-entity'
};
db.getItem(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    if (data.Item) {
      console.log('item', data.Item);
    } else {
      console.error('no item found');
    }
  }
});
```

Specifies the attributes of the primary key

Invokes the getItem operation on DynamoDB

Checks whether an item was found

The `user <uid>` command retrieves a user by the user's ID. The code to fetch a user is shown in listing 12.8.

Listing 12.8 nodetodo: Retrieving a user (index.js)

```
const mapUserItem = (item) => {
  return {
    uid: item.uid.S,
    email: item.email.S,
    phone: item.phone.S
  };
};

if (input['user'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}
    },
    TableName: 'todo-user'
  };
  db.getItem(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      if (data.Item) {
        console.log('user', mapUserItem(data.Item));
      } else {
        console.error('user not found');
      }
    }
  });
}
```

Helper function to transform DynamoDB result

Looks up a user by primary key

Specifies the user table

Invokes the getItem operation on DynamoDB

Checks whether data was found for the primary key

For example, use the following command to fetch information about Emma:

```
node index.js user emma
```

You can also use the `getItem` operation to retrieve data by partition key and sort key, for example, to look up a specific task. The only change is that the `Key` has two entries instead of one. `getItem` returns one item or no items; if you want to get multiple items, you need to query DynamoDB.

12.4.2 Querying items by key and filter

Emma wants to get check her to-do list for things she needs to do. Therefore, you will query the `todo-task` table to retrieve all tasks assigned to Emma next.

If you want to retrieve a collection of items rather than a single item, such as all tasks for a user, you must query DynamoDB. Retrieving multiple items by primary key works only if your table has a partition key and sort key. Otherwise, the partition key will identify only a single item. Listing 12.9 shows how the `query` SDK operation can be used to get a collection of items from DynamoDB.

Listing 12.9 DynamoDB: Querying a table

```
const params = {
  KeyConditionExpression: 'attr1 = :attr1val
➥ AND attr2 = :attr2val',
  ExpressionAttributeValues: {
    ':attr1val': {S: 'val1'},
    ':attr2val': {N: '2'}
  },
  TableName: 'app-entity'
};
db.query(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
  }
});
```

The condition the key must match. Use AND if you're querying both a partition and sort key. Only the = operator is allowed for partition keys. Allowed operators for sort keys are =, >, <, >=, <=, BETWEEN x AND y, and begins_with. Sort key operators are blazing fast because the data is already sorted.

Dynamic values are referenced in the expression.

Always specify the correct type (S, N, B).

Invokes the query operation on DynamoDB

The query operation also lets you specify an optional `FilterExpression`, to include only items that match the filter and key condition. This is helpful to reduce the result set, for example, to show only tasks of a specific category. The syntax of `FilterExpression` works like `KeyConditionExpression`, but no index is used for filters. Filters are applied to all matches that `KeyConditionExpression` returns.

To list all tasks for a certain user, you must query DynamoDB. The primary key of a task is the combination of the `uid` and the `tid`. To get all tasks for a user, `KeyConditionExpression` requires only the partition key.

The next listing shows two helper functions used to implement the `task-ls` command.

Listing 12.10 nodetodo: Retrieving tasks (index.js)

```
const getValue = (attribute, type) => {
  if (attribute === undefined) {
    return null;
  }
  return attribute[type];
};

const mapTaskItem = (item) => {
  return {
    tid: item.tid.N,
    description: item.description.S,
    created: item.created.N,
    due: getValue(item.due, 'N'),
    category: getValue(item.category, 'S'),
    completed: getValue(item.completed, 'N')
  };
};
```

Helper function to access optional attributes

Helper function to transform the DynamoDB result

The real magic happens in listing 12.11, which shows the implementation of the `task-ls` command.

Listing 12.11 nodetodo: Retrieving tasks (index.js)

```
if (input['task-ls'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    KeyConditionExpression: 'uid = :uid',
    ExpressionAttributeValues: {
      ':uid': {S: input['<uid>']}
    },
    TableName: 'todo-task',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression += ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues
➥ [':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--due'] === true) {
    params.FilterExpression = 'due = :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--withoutdue'] === true) {
    params.FilterExpression =
➥ 'attribute_not_exists(due)';
  } else if (input['--futuredue'] === true) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--dueafter'] !== null) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
      {N: input['--dueafter']};
  } else if (input['--duebefore'] !== null) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
      {N: input['--duebefore']};
  }
  if (input['<category>'] !== null) {
    if (params.FilterExpression === undefined) {
      params.FilterExpression = '';
    } else {
      params.FilterExpression += ' AND ';
    }
    params.FilterExpression += 'category = :category';
    params.ExpressionAttributeValues[':category'] =
      S: input['<category>']};
  }
  db.query(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next=' +
```

Primary key query. The task table uses a partition and sort key. Only the partition key is defined in the query, so all tasks belonging to a user are returned.

Query attributes must be passed this way.

Filter attributes must be passed this way.

Filtering uses no index; it's applied over all elements returned from the primary key query.

attribute_not_exists(due) is true when the attribute is missing (opposite of attribute_exists).

Multiple filters can be combined with logical operators.

Invokes the query operation on DynamoDB

```
            data.LastEvaluatedKey.tid.N);
        }
      }
    });
}
```

As an example, you could use the following command to get a list of Emma's shopping tasks. The query will fetch all tasks by `uid` first and filter items based on the category attribute next:.

```
node index.js task-ls emma shopping
```

Two problems arise with the query approach:

- Depending on the result size from the primary key query, filtering may be slow. Filters work without an index: every item must be inspected. Imagine you have stock prices in DynamoDB, with a partition key and sort key: the partition key is a ticker like AAPL, and the sort key is a timestamp. You can make a query to retrieve all stock prices of Apple (AAPL) between two timestamps (20100101 and 20150101). But if you want to return prices only on Mondays, you need to filter over all prices to return only 20% (one out of five trading days each week) of them. That's wasting a lot of resources!
- You can only query the primary key. Returning a list of all tasks that belong to a certain category for all users isn't possible, because you can't query the `category` attribute.

You can solve these problems with secondary indexes. Let's look at how they work.

12.4.3 Using global secondary indexes for more flexible queries

In this section, you will create a secondary index that allows you to query the tasks belonging to a certain category.

A *global secondary index* is a projection of your original table that is automatically maintained by DynamoDB. Items in an index don't have a primary key, just a key. This key is not necessarily unique within the index. Imagine a table of users where each user has a country attribute. You then create a global secondary index where the country is the new partition key. As you can see, many users can live in the same country, so that key is not unique in the index.

You can query a global secondary index like you would query the table. You can imagine a global secondary index as a read-only DynamoDB table that is automatically maintained by DynamoDB: whenever you change the parent table, all indexes are asynchronously (eventually consistent!) updated as well.

In our example, we will create a global secondary index for the table `todo-tasks` which uses the `category` as the partition key and the `tid` as the sort key. Doing so allows us to query tasks by category. Figure 12.4 shows how a global secondary index works.

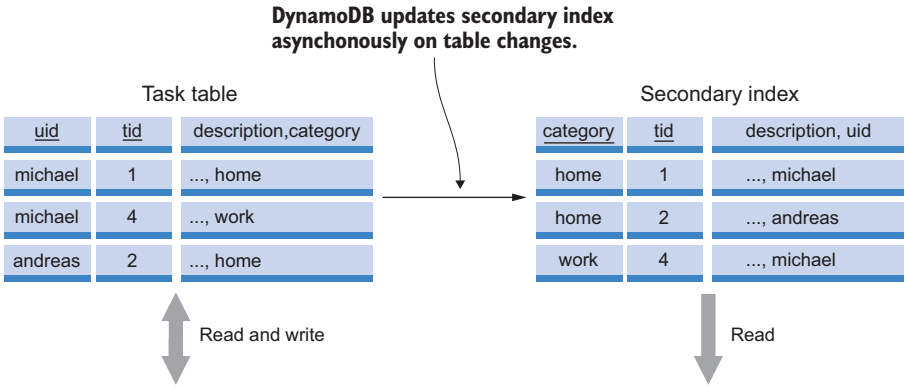


Figure 12.4 A global secondary index contains a copy (projection) of your table's data to provide fast lookup on another key.

A global secondary index comes at a price: the index requires storage (the same cost as for the original table). You must provision additional write-capacity units for the index as well, because a write to your table will cause a write to the global secondary index as well.

> **Local secondary index**
>
> Besides global secondary indexes, DynamoDB also supports local secondary indexes. A local secondary index must use the same partition key as the table. You can only vary on the attribute that is used as the sort key. A local secondary index consumes the read and write capacity of the table.

A huge benefit of DynamoDB is that you can provision capacity based on your workload. If one of your global secondary indexes gets tons of read traffic, you can increase the read capacity of that index. You can fine-tune your database performance by provisioning sufficient capacity for your tables and indexes. You'll learn more about that in section 13.9.

12.4.4 Creating and querying a global secondary index

Back to nodetodo. John needs the shopping list including Emma's and his tasks for his trip to town.

To implement the retrieval of tasks by category, you'll add a secondary index to the `todo-task` table. This will allow you to make queries by category. A partition key and sort key are used: the partition key is the `category` attribute, and the sort key is the `tid` attribute. The index also needs a name: `category-index`. You can find the following CLI command in the README.md file in nodetodo's code folder and simply copy and paste:

```
$ aws dynamodb update-table --table-name todo-task \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ AttributeName=category,AttributeType=S \
➥ --global-secondary-index-updates '[{\
➥ "Create": {\
➥ "IndexName": "category-index", \
➥ "KeySchema": [{"AttributeName": "category",
➥ "KeyType": "HASH"}, \
➥ {"AttributeName": "tid", "KeyType": "RANGE"}], \
➥ "Projection": {"ProjectionType": "ALL"}, \
➥ "ProvisionedThroughput": {"ReadCapacityUnits": 5, \
➥ "WriteCapacityUnits": 5}\
➥ }}]'
```

Creates a new secondary index

Adds a global secondary index by updating the table that has already been created previously

Adds a category attribute, because the attribute will be used in the index

The category attribute is the partition key, and the tid attribute is the sort key.

All attributes are projected into the index.

Creating a global secondary index takes about five minutes. You can use the CLI to find out if the index is ready like this:

```
$ aws dynamodb describe-table --table-name=todo-task \
➥ --query "Table.GlobalSecondaryIndexes"
```

The next listing shows the code to query the global secondary index to fetch tasks by category with the help of the `task-la` command.

Listing 12.12 nodetodo: Retrieving tasks from a global secondary index (index.js)

```
if (input['task-la'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    KeyConditionExpression: 'category = :category',
    ExpressionAttributeValues: {
      ':category': {S: input['<category>']}
    },
    TableName: 'todo-task',
    IndexName: 'category-index',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression += ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  }
  [...]
  db.query(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
```

A query against an index works the same as a query against a table …

… but you must specify the index you want to use.

Filtering works the same as with tables.

```
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next='
          + data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

When following our example, use the following command to fetch John and Emma's shopping tasks:

```
node index.js task-la shopping
```

But you'll still have situations where a query doesn't work. For example, you can't retrieve all users from `todo-user` with a query. Therefore, let's look at how to scan through all items of a table.

12.4.5 Scanning and filtering all of your table's data

John wants to know who else is using nodetodo and asks for a list of all users. You will learn how to list all items of a table without using an index next.

Sometime you can't work with keys because you don't know them up front; instead, you need to go through all the items in a table. That's not very efficient, but in rare situations, like daily batch jobs or rare requests, it's fine. DynamoDB provides the `scan` operation to scan all items in a table as shown next.

Listing 12.13 DynamoDB: Scan through all items in a table

```
const params = {
  TableName: 'app-entity',
  Limit: 50
};
db.scan(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
    if (data.LastEvaluatedKey !== undefined) {
      console.log('more items available');
    }
  }
});
```

Specifies the maximum number of items to return

Invokes the scan operation on DynamoDB

Checks whether there are more items that can be scanned

The following listing shows how to scan through all items of the `todo-user` table. A paging mechanism is used to prevent too many items from being returned at a time.

Listing 12.14 nodetodo: Retrieving all users with paging (index.js)

```
if (input['user-ls'] === true) {
  const params = {
    TableName: 'todo-user',
```

```
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.ExclusiveStartKey = {
      uid: {S: input['--next']}
    };
  }
  db.scan(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('users', data.Items.map(mapUserItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('page with --next=' + data.LastEvaluatedKey.uid.S);
      }
    }
  });
}
```

The maximum number of items returned

The named parameter next contains the last evaluated key.

Invokes the scan operation on DynamoDB

Checks whether the last item has been reached

Use the following command to fetch all users:

```
node index.js user-ls
```

The `scan` operation reads all items in the table. This example didn't filter any data, but you can use `FilterExpression` as well. Note that you shouldn't use the `scan` operation too often—it's flexible but not efficient.

12.4.6 Eventually consistent data retrieval

By default, reading data from DynamoDB is *eventually consistent*. That means it's possible that if you create an item (version 1), update that item (version 2), and then read that item, you may get back version 1; if you wait and fetch the item again, you'll see version 2. Figure 12.5 shows this process. This behavior occurs because the item is

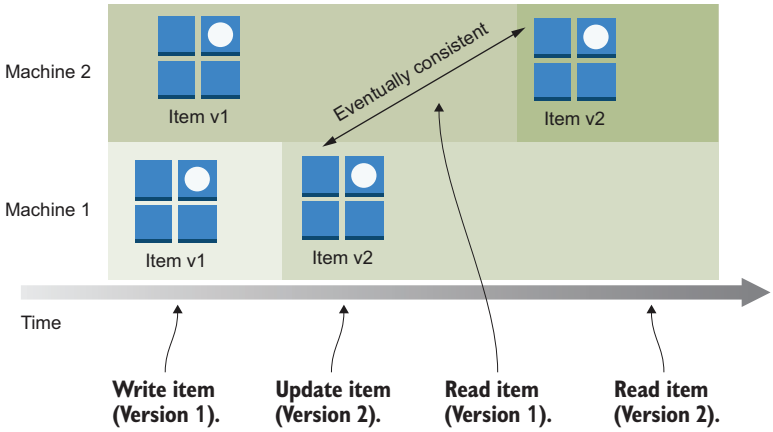


Figure 12.5 Eventually consistent reads can return old values after a write operation until the change is propagated to all machines.

persisted on multiple machines in the background. Depending on which machine answers your request, the machine may not have the latest version of the item.

You can prevent eventually consistent reads by adding `"ConsistentRead": true` to the DynamoDB request to get *strongly consistent reads*. Strongly consistent reads are supported by `getItem`, `query`, and `scan` operations. But a strongly consistent read is more expensive—it takes longer and consumes more read capacity—than an eventually consistent read. Reads from a global secondary index are always eventually consistent because the synchronization between table and index happens asynchronously.

Typically, NoSQL databases do not support transactions with atomicity, consistency, isolation, and durability (ACID) guarantees. However, DynamoDB comes with the ability to bundle multiple read or write requests into a transaction. The relevant API methods are called `TransactWriteItems` and `TransactGetItems`—you can either group write or read requests, but not a mix of both of them. Whenever possible, you should avoid using transactions because they are more expensive from a cost and latency perspective. Check out http://mng.bz/E0ol if you are interested in the details about DynamoDB transactions.

12.5 Removing data

John stumbled across a fancy to-do application on the web. Therefore, he decides to remove his user from nodetodo.

Like the `getItem` operation, the `deleteItem` operation requires that you specify the primary key you want to delete. Depending on whether your table uses a partition key or a partition key and sort key, you must specify one or two attributes. The next listing shows the implementation of the `user-rm` command.

Listing 12.15 nodetodo: Removing a user (index.js)

```
if (input['user-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}
    },
    TableName: 'todo-user'
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('user removed');
    }
  });
}
```

Identifies an item by partition key

Specifies the user table

Invokes the deleteItem operation on DynamoDB

Use the following command to delete John from the DynamoDB table:

```
node index.js user-rm john
```

But John wants to delete not only his user but also his tasks. Luckily, removing a task works similarly. The only change is that the item is identified by a partition key and sort key and the table name has to be changed. The code used for the `task-rm` command is shown here.

Listing 12.16 nodetodo: Removing a task (index.js)

```
if (input['task-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}
    },
    TableName: 'todo-task'
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task removed');
    }
  });
}
```

Identifies an item by partition key and sort key

Specifies the task table

You're now able to create, read, and delete items in DynamoDB. The only operation you're missing is updating an item.

12.6 Modifying data

Emma is still a big fan of nodetodo and uses the application constantly. She just bought some milk and wants to mark the task as done.

You can update an item with the `updateItem` operation. You must identify the item you want to update by its primary key; you can also provide an `UpdateExpression` to specify the updates you want to perform. Use one or a combination of the following update actions:

- Use `SET` to override or create a new attribute. Examples: `SET attr1 = :attr1val`, `SET attr1 = attr2 + :attr2val`, `SET attr1 = :attr1val, attr2 = :attr2val`.
- Use `REMOVE` to remove an attribute. Examples: `REMOVE attr1`, `REMOVE attr1, attr2`.

To implement this feature, you need to update the task item, as shown next.

Listing 12.17 nodetodo: Updating a task as done (index.js)

```
if (input['task-done'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}
    },
```

Identifies the item by a partition and sort key

```
    UpdateExpression: 'SET completed = :yyyymmdd',
    ExpressionAttributeValues: {
      ':yyyymmdd': {N: yyyymmdd}
    },
    TableName: 'todo-task'
  };
  db.updateItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task completed');
    }
  });
}
```

Defines which attributes should be updated

Attribute values must be passed this way.

Invokes the updateItem operation on DynamoDB

For example, the following command closes Emma's task to buy milk. Please note: the `<tid>` will differ when following the example yourself. Use `node index.js task-ls emma` to get the task's ID:

```
node index.js task-done emma 1643037541999
```

12.7 Recap primary key

We'd like to recap an important aspect of DynamoDB, the primary key. A primary key is unique within a table and identifies an item. You can use a single attribute as the primary key. DynamoDB calls this a partition key. You need an item's partition key to look up that item. Also, when updating or deleting an item, DynamoDB requires the partition key. You can also use two attributes as the primary key. In this case, one of the attributes is the partition key, and the other is called the sort key.

12.7.1 Partition key

A partition key uses a single attribute of an item to create a hash-based index. If you want to look up an item based on its partition key, you need to know the exact partition key. For example, a user table could use the user's email address as a partition key. The user could then be retrieved if you know the partition key—the email address, in this case.

12.7.2 Partition key and sort key

When you use both a partition key and a sort key, you're using two attributes of an item to create a more powerful index. To look up an item, you need to know its exact partition key, but you don't need to know the sort key. You can even have multiple items with the same partition key: they will be sorted according to their sort key.

The partition key can be queried only using exact matches (`=`). The sort key can be queried using `=`, `>`, `<`, `>=`, `<=`, and `BETWEEN x AND y` operators. For example, you can query the sort key of a partition key from a certain starting point. You cannot query only the sort key—you must always specify the partition key. A message table could use a partition key and sort key as its primary key; the partition key could be the user's

email, and the sort key could be a timestamp. You could then look up all of user's messages that are newer or older than a specific timestamp, and the items would be sorted according to the timestamp.

12.8 SQL-like queries with PartiQL

As the developer of nodetodo, we want to get some insight into the way our users use the application. Therefore, we are looking for a flexible approach to query the data stored on DynamoDB on the fly.

Because SQL is such a widely used language, hardly any database system can avoid offering an SQL interface as well, even if, in the case of NoSQL databases, often only a fraction of the language is supported. In the following examples, you will learn how to use PartiQL, which is designed to provide unified query access to all kinds of data and data stored. DynamoDB supports PartiQL via the Management Console, NoSQL Workbench, AWS CLI, and DynamoDB APIs. Be aware that DynamoDB supports only a small subset of the PartiQL language. The following query lists all tasks stored in table `todo-task`:

The command execute-statement supports PartiQL statements as well.

```
$ aws dynamo3db execute-statement \
➥ --statement "SELECT * FROM \""todo-task\""" 
```

A simple SELECT statement to fetch all attributes of all items from table todo-task. The escaped " is required because the table name includes a hyphen.

PartiQL allows you to query an index as well. The following statement fetches all tasks with category equals `shopping` from index `category-index`:

```
$ aws dynamodb execute-statement --statement \
➥ "SELECT * FROM \""todo-task\".\"category-index\""
➥ WHERE category = 'shopping'"
```

Please note that combining multiple queries (`JOIN`) is not supported by DynamoDB. However, DynamoDB supports modifying data with the help of PartiQL. The following command updates Emma's phone number:

```
aws dynamodb execute-statement --statement \
➥ "Update \"todo-user\" SET phone='+33333333' WHERE uid='emma'"
```

Be warned, an `UPDATE` or `DELETE` statement must include a `WHERE` clause that identifies a single item by its partition key or partition and sort keys. Therefore, it is not possible to update or delete more than one item per query.

Want to learn more about PartiQL for DynamoDB? Check out the official documentation: http://mng.bz/N5g2.

In our opinion, using PartiQL is confusing because it pretends to provide a flexible SQL language but is in fact very limited. We prefer using the DynamoDB APIs and the SDK, which is much more descriptive.

12.9 DynamoDB Local

Imagine a team of developers is working on a new app using DynamoDB. During development, each developer needs an isolated database so as not to corrupt the other team members' data. They also want to write unit tests to make sure their app is working. To address their needs, you could create a unique set of DynamoDB tables with a CloudFormation stack for each developer. Or you could use a local DynamoDB for offline development. AWS provides a local implementation of DynamoDB, which is available for download at http://mng.bz/71qm.

Don't run DynamoDB Local in production! It's only made for development purposes and provides the same functionality as DynamoDB, but it uses a different implementation: only the API is the same.

> **NoSQL Workbench for DynamoDB**
> Are you looking for a graphical user interface to interact with DynamoDB? Check out NoSQL Workbench for DynamoDB at http://mng.bz/mJ5P. The tool allows you to create data models, analyze data, and import and export data.

That's it! You've implemented all of nodetodo's features.

12.10 Operating DynamoDB

DynamoDB doesn't require administration like a traditional relational database, because it's a managed service and AWS takes care of that; instead, you only have a few things to do.

With DynamoDB, you don't need to worry about installation, updates, machines, storage, or backups. Here's why:

- DynamoDB isn't software you can download. Instead, it's a NoSQL database as a service. Therefore, you can't install DynamoDB like you would MySQL or MongoDB. This also means you don't have to update your database; the software is maintained by AWS.
- DynamoDB runs on a fleet of machines operated by AWS. AWS takes care of the OS and all security-related questions. From a security perspective, it's your job to restrict access to your data with the help of IAM.
- DynamoDB replicates your data among multiple machines and across multiple data centers. There is no need for a backup from a durability point of view. However, you should configure snapshots to be able to recover from accidental data deletion.

Now you know some administrative tasks that are no longer necessary if you use DynamoDB. But you still have things to consider when using DynamoDB in production:

monitoring capacity usage, provisioning read and write capacity (section 12.11), and creating backups of your tables.

Backups

DynamoDB provides very high durability. But what if the database administrator accidentally deletes all the data or a new version of the application corrupts items? In this case, you would need a backup to restore to a working table state from the past. In December 2017, AWS announced a new feature for DynamoDB: on-demand backup and restore. We strongly recommend using on-demand backups to create snapshots of your DynamoDB tables to be able to restore them later, if needed.

12.11 Scaling capacity and pricing

As discussed at the beginning of the chapter, the difference between a typical relational database and a NoSQL database is that a NoSQL database can be scaled by adding new nodes. Horizontal scaling enables increasing the read and write capacity of a database enormously. That's the case for DynamoDB as well. However, a DynamoDB table has two different read/write capacity modes:

- *On-demand mode* adapts the read and write capacity automatically.
- *Provisioned mode* requires you to configure the read and write capacity upfront. Additionally, you have the option to enable autoscaling to adapt the provisioned capacity based on the current load automatically.

At first glance, the first option sounds much better because it is totally maintenance free. Table 12.1 compares the costs between on-demand and provisioned modes. All of the following examples are based on costs for `us-east-1` and assume you are reading and writing items with less than 1 KB.

Table 12.1 Comparing pricing of DynamoDB on-demand and provisioned mode

Throughput	On-demand mode	Provisioned mode
10 writes per second	$32.85 per month	$4.68 per month
100 reads per second	$32.85 per month	$4.68 per month

That's a significant difference. Accessing an on-demand table costs about seven times as much as using a table in provisioned capacity mode. But the example is not significant, because of the assumption that the provisioned throughput of 10 writes and 100 reads per second is running at 100% capacity 24/7. This is hardly true for any application. Most applications have large variations in read and write throughput. Therefore, in many cases, the opposite is correct: on-demand is more cost-efficient than provisioned. For example, we are operating a chatbot called marbot on AWS. When we switched from provisioned to on-demand capacity in December 2018, we reduced our monthly costs for DynamoDB by 90% (see http://mng.bz/DD69 for details).

As a rule of thumb, the more spikes in your traffic, the more likely that on-demand is the best option for you. Let's illustrate this with another example. Assume your application is used only during business hours from 9 a.m. to 5 p.m. The baseline throughput is 100 reads and 10 write per second. However, a batch job is running between 4 p.m. and 5 p.m., which requires 1,000 reads and 100 writes per second.

With provisioned capacity mode, you need to provision for the peak load. That's 1,000 reads and 10 writes per second. That's why the monthly costs for a table with provisioned capacity costs is $24.75 higher than when using on-demand capacity in this example, as shown here:

```
// Provisioned capacity mode
$0.00065 for 1 writes/sec * 100 * 24 hours * 30 days = $46.80 per month
$0.00013 for 2 reads/sec * 500 * 24 hours * 30 days = $46.80 per month

Read/Writes with provisioned capacity mode per month: $93.60

// On-demand Capacity Mode
(7 hours * 60 min * 60 sec * 10 write) + (1 hour * 60 min * 60 sec * 100
     writes) = 612,000 writes/day
(7 hours * 60 min * 60 sec * 100 reads) + (1 hour * 60 min * 60 sec * 1000
     reads) = 6,120,000 reads/day

$0.00000125/write * 612,000 writes/day * 30 days = $22.95
$0.00000025/read * 6,120,000 reads/day * 30 days = $45.90

Read/Writes with On-demand Mode per month: $68.85
```

By the way, you are not only paying for accessing your data but also for storage. By default, you are paying $0.25 per GB per month. The first 25 GB are free. You only pay per use—there is no need to provision data upfront.

Does your workload require storing huge amounts of data that are accessed rarely? If so, you should have a look into Standard-Infrequent Access, which reduces the cost for storing data to $0.10 per GB per month but increases costs for reading and writing data by about 25%. Please visit the https://aws.amazon.com/dynamodb/pricing/ for more details on DynamoDB pricing.

12.11.1 Capacity units

When working with provisioned capacity mode, you need to configure the provisioned read and write capacity separately. To understand capacity units, let's start by experimenting with the CLI, as shown here:

```
$ aws dynamodb get-item --table-name todo-user \
➥ --key '{"uid": {"S": "emma"}}' \
➥ --return-consumed-capacity TOTAL \
➥ --query "ConsumedCapacity"
{
    "CapacityUnits": 0.5,
    "TableName": "todo-user"
}
```

Tells DynamoDB to return the used capacity units

getItem requires 0.5 capacity units.

```
$ aws dynamodb get-item --table-name todo-user \
➥ --key '{"uid": {"S": "emma"}}' \
➥ --consistent-read --return-consumed-capacity TOTAL \
➥ --query "ConsumedCapacity"
{
    "CapacityUnits": 1.0,
    "TableName": "todo-user"
}
```

A consistent read …

… needs twice as many capacity units.

More abstract rules for throughput consumption follow:

- An eventually consistent read takes half the capacity of a strongly consistent read.
- A strongly consistent `getItem` requires one read capacity unit if the item isn't larger than 4 KB. If the item is larger than 4 KB, you need additional read capacity units. You can calculate the required read capacity units using `roundUP(itemSize / 4)`.
- A strongly consistent `query` requires one read capacity unit per 4 KB of item size. This means if your query returns 10 items, and each item is 2 KB, the item size is 20 KB and you need five read units. This is in contrast to 10 `getItem` operations, for which you would need 10 read capacity units.
- A write operation needs one write capacity unit per 1 KB of item size. If your item is larger than 1 KB, you can calculate the required write capacity units using `roundUP(itemSize)`.

If capacity units aren't your favorite unit, you can use the AWS Pricing Calculator at https://calculator.aws to calculate your capacity needs by providing details of your read and write workload.

The provision throughput of a table or a global secondary index is defined in seconds. If you provision five read capacity units per second with `ReadCapacityUnits=5`, you can make five strongly consistent `getItem` requests for that table if the item size isn't larger than 4 KB per second. If you make more requests than are provisioned, DynamoDB will first throttle your request. If you make many more requests than are provisioned, DynamoDB will reject your requests.

Increasing the provisioned throughput is possible whenever you like, but you can only decrease the throughput of a table four to 23 times a day (a day in UTC time). Therefore, you might need to overprovision the throughput of a table during some times of the day.

Limits for decreasing the throughput capacity

Decreasing the throughput capacity of a table is generally allowed only four times a day (a day in UTC time). Additionally, decreasing the throughout capacity is possible even if you have used up all four decreases if the last decrease has happened more than an hour ago.

It is possible but not necessary to update the provisioned capacity manually. By using Application Auto Scaling, you can increase or decrease the capacity of your table or global secondary indices based on a CloudWatch metric automatically. See http://mng.bz/lR7M to learn more.

12.12 Networking

DynamoDB does not run in your VPC. It is accessible via an API. You need internet connectivity to reach the DynamoDB API. This means you can't access DynamoDB from a private subnet by default, because a private subnet has no route to the internet via an internet gateway. Instead, a NAT gateway is used (see section 5.5 for more details). Keep in mind that an application using DynamoDB can create a lot of traffic, and your NAT gateway is limited to 10 Gbps of bandwidth. A better approach is to set up a VPC endpoint for DynamoDB and use that to access DynamoDB from private subnets without needing a NAT gateway at all. You can read more about VPC endpoints in the AWS documentation at http://mng.bz/51qz.

12.13 Comparing DynamoDB to RDS

In chapter 10, you learned about the Relational Database Service (RDS). For a better understanding, let's compare the two different database systems. Table 12.2 compares DynamoDB and the RDS. Keep in mind that this is like comparing apples and oranges: the only thing DynamoDB and RDS have in common is that both are called databases. RDS provides relational databases that are very flexible when it comes to ingesting or querying data. However, scaling RDS is a challenge. Also, RDS is priced per database instance hour. In contrast, DynamoDB scales horizontally with little or no effort. Also, DynamoDB offers a pay-per-request pricing model, which is interesting for low-volume workloads with traffic spikes. Use RDS if your application requires complex SQL queries or you don't want to invest time into mastering a new technology. Otherwise, you can consider migrating your application to DynamoDB.

Table 12.2 Differences between DynamoDB and RDS

Task	DynamoDB	RDS Aurora
Creating a table	Management Console, SDK, or CLI `aws dynamodb create-table`	SQL `CREATE TABLE` statement
Inserting, updating, or deleting data	SDK or PartiQL (limited version of SQL)	SQL `INSERT`, `UPDATE`, or `DELETE` statement
Querying data	SDK `query` or PartiQL	SQL `SELECT` statement
Adding indexes to extend the possibility of query data	No more than 25 global secondary indexes per table	Number of indexes not limited per table

Table 12.2 Differences between DynamoDB and RDS *(continued)*

Task	DynamoDB	RDS Aurora
Increasing storage	No action needed; storage grows and shrinks automatically.	Increasing storage is possible via the Management Console, CLI, or API.
Increasing performance	Horizontal, by increasing capacity. DynamoDB will add more machines under the hood.	Vertical, by increasing instance size and disk throughput; or horizontal, by adding up to five read replicas
Distribute data globally	Multiactive replication enables reads and writes in multiple regions.	Read replication enables data synchronization to other regions as well, but writing data is possible only in the source region.
Installing the database on your machine	DynamoDB is available on AWS only. There is a local version for developing on your local machine.	Install MySQL or PostgreSQL on your machine.
Hiring an expert	DynamoDB skills needed	General SQL skills sufficient for most scenarios

12.14 NoSQL alternatives

Besides DynamoDB, a few other NoSQL options are available on AWS as shown in table 12.3. Make sure you understand the requirements of your workload and learn about the details of a NoSQL database before deciding on an option.

Table 12.3 Differences between DynamoDB and some NoSQL databases

Amazon Keyspaces	Columnar store	A fully managed Apache Cassandra–compatible database service	A good fit when dealing with very large data sets. Think of DynamoDB as an open source project. Available as a managed service by other vendors as well.
Amazon Neptune	Graph store	A proprietary graph database provided by AWS	Perfect fit for a social graph, personalization, product catalog, highly connected data sets
Amazon DocumentDB with MongoDB compatibility	Document store	A database service that is purpose-built for JSON data management at scale, fully managed and integrated with AWS. Amazon DocumentDB is compatible with MongoDB 3.6.	A good choice if you are looking for a NoSQL database that also brings flexible query capabilities. A common use case are common CRUD (create, read, update, and delete) applications.
Amazon MemoryDB for Redis	Key-value store	An in-memory database with durability, providing a Redis-compatible interface	A good match for implementing a cache, a session store, or whenever I/O latency is super critical

Cleaning up

Don't forget to delete your DynamoDB tables after you finish this chapter like so:

```
aws dynamodb delete-table --table-name todo-task
aws dynamodb delete-table --table-name todo-user
```

Summary

- DynamoDB is a NoSQL database service that removes all the operational burdens from you, scales well, and can be used in many ways as the storage backend of your applications.
- Looking up data in DynamoDB is based on keys. To query an item, you need to know the primary key, called the partition key.
- When using a combination of partition key and sort key, many items can use the same partition key as long as their sort key does not overlap. This approach also allows you to query all items with the same partition key, ordered by the sort key.
- Adding a global secondary index allows you to query efficiently on an additional attribute.
- The `query` operation queries a table or secondary indexes.
- The `scan` operation searches through all items of a table, which is flexible but not efficient and shouldn't be used too extensively.
- Enforcing strongly consistent reads avoids running into eventual consistency problems with stale data. But reading from a global secondary index is always eventually consistent.
- DynamDB comes with two capacity modes: on demand and provisioned. The on-demand capacity mode works best for spiky workloads.