5 Securing your system: IAM, security groups, and VPC

This chapter covers

- Who is responsible for security?
- Keeping your software up-to-date
- Controlling access to your AWS account with users and roles
- Keeping your traffic under control with security groups
- Using CloudFormation to create a private network

If security is a wall, you'll need a lot of bricks to build that wall, as shown in figure 5.1. This chapter focuses on the following four most important bricks to secure your systems on AWS:

1. *Installing software updates*—New security vulnerabilities are found in software every day. Software vendors release updates to fix those vulnerabilities, and it's your job to install those updates as quickly as possible after they're released on your systems. Otherwise, your systems will be an easy victim for hackers.
2. *Restricting access to your AWS account*—This becomes even more important if you aren't the only one accessing your AWS account, such as when coworkers and automated processes need access to your AWS account as well. A buggy script could easily terminate all your EC2 instances instead of only the one you intended. Granting only the permissions needed is the key to securing your AWS resources from accidental or intended disastrous actions.
3. *Controlling network traffic to and from your EC2 instances*—You want ports to be accessible only if they must be. If you run a web server, the only ports you need to open to the outside world are ports 80 for HTTP traffic and 443 for HTTPS traffic. Do not open any other ports for external access.
4. *Creating a private network in AWS*—Control network traffic by defining subnets and routing tables. Doing so allows you to specify private networks that are not reachable from the outside.

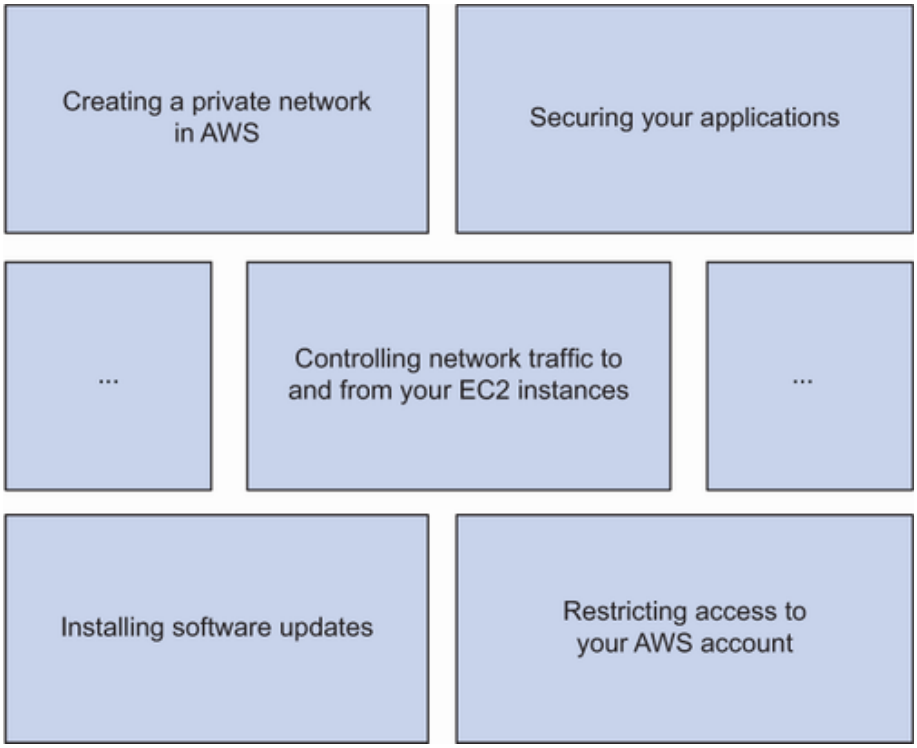

Figure 5.1 To achieve the security of your cloud infrastructure and application, all security building blocks have to be in place.

One important brick is missing: securing your applications. We do not cover application security in our book. When buying or developing applications, you should follow security standards. For example, you need to check user input and allow only the necessary characters, don't save passwords in plain text, and use TLS/SSL to encrypt traffic between your virtual machines and your users.

This is going to be a long chapter—security is such an important topic, there's a lot to cover. But don't worry, we'll take it step by step.

Not all examples are covered by Free Tier

The examples in this chapter are not all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them 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.

Chapter requirements

To fully understand this chapter, you should be familiar with the following networking concepts:

- Subnet
- Route tables
- Access control lists (ACLs)
- Gateway
- Firewall
- Port
- Access management
- Basics of the Internet Protocol (IP), including IP addresses

Before we look at the four bricks, let's talk about how responsibility is divided between you and AWS.

5.1 Who's responsible for security?

The cloud is a shared-responsibility environment, meaning responsibility is shared between you and AWS. AWS is responsible for the following:

- Protecting the network through automated monitoring systems and robust internet access, to prevent distributed denial of service (DDoS) attacks
- Performing background checks on employees who have access to sensitive areas
- Decommissioning storage devices by physically destroying them after end of life
- Ensuring the physical and environmental security of data centers, including fire protection and security staff

The security standards are reviewed by third parties; you can find an up-to-date overview at **https://aws.amazon.com/compliance/**.

What are your responsibilities? See the following:

- Configuring access management that restricts access to AWS resources like S3 and EC2 to a minimum, using AWS IAM
- Encrypting network traffic to prevent attackers from reading or manipulating data (e.g., using HTTPS)
- Configuring a firewall for your virtual network that controls incoming and outgoing traffic with security groups and NACLs
- Encrypting data at rest. For example, enable data encryption for your database or other storage systems
- Managing patches for the OS and additional software on virtual machines

Security involves an interaction between AWS and you, the customer. If you play by the rules, you can achieve high security standards in the

cloud. Want to dive into more details? Check out **https://aws.amazon.-com/compliance/shared-responsibility-model/**.

5.2 Keeping the operating system up-to-date

Not a week goes by without the release of an important update to fix security vulnerabilities in some piece of software or another. Sometimes the kernel is affected or libraries, like OpenSSL. Other times, it's affecting an environment like Java, Apache, and PHP, or an application like WordPress. If a security update is released, you must install it quickly, because the exploit may have already been released, or because unscrupulous people could look at the source code to reconstruct the vulnerability. You should have a working plan for how to apply updates to all running virtual machines as quickly as possible.

Amazon Linux 2 installs critical or important security updates automatically on startup while `cloud-init` is running. We highly recommend you install all the other updates as well. The following options are available:

- *Install all updates at the end of the boot process*—Include `yum -y update` in your user-data script. `yum` is the package manager used by Amazon Linux 2.
- *Install security updates at the end of the boot process only*—Include the `yum -y --security update` in your user-data script.
- Use the *AWS Systems Manager Patch Manager*—Install updates based on a patch baseline.

The first two options can be easily included in the user data of your EC2 instance. You can find the code in /chapter05/ec2-yum-update.yaml in the book's code folder. You install all updates as follows:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    UserData: !Base64 |
      #!/bin/bash -ex
      yum -y update    ①
```

① Installs all updates

To install only security updates, do the following:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    UserData: !Base64 |
      #!/bin/bash -ex
      yum -y --security update    ①
```

① Installs only security updates

The following challenges are still waiting for a solution:

- The problem with installing all updates is that your system might still be vulnerable. Some updates require a reboot (most notably kernel updates)!
- Installing updates on startup is not enough. Updates need to be installed continuously.

Before reinventing the wheel, it is a good strategy to research whether AWS provides the building blocks needed to get the job done. Luckily, AWS Systems Manager (SSM) Patch Manager is a good choice to make patching more robust and stable.

The AWS Systems Manager provides a toolbox that includes a core set of features bundled into capabilities. Patch Manager is one such capability. The following core SSM features are bundled together in the Patch Manager, as figure 5.2 shows:

- *Agent*—Preinstalled and autostarted on Amazon Linux 2 (also powers the Session Manager).
- *Document*—Think of a document as a script on steroids. We use a prebuild document named `AWS-RunPatchBaseline` to install patches.
- *Run Command*—Executes a document on an EC2 instance.
- *Association*—Sends commands (via Run Command) to EC2 instances on a schedule or during startup (bundled into the capability named State Manager).
- *Maintenance Window*—Sends commands (via Run Command) to EC2 instances on a schedule during a time window.

- *Patch baseline*—Set of rules to approve patches for installation based on classification and severity. Luckily, AWS provides predefined patch baselines for various operating systems including Amazon Linux 2. The predefined patch baseline for Amazon Linux 2 approves all security patches that have a severity level of critical or important and all bug fixes. A seven-day waiting period exists after the release of a patch before approval.

Figure 5.2 SSM features required for Patch Manager capability

The following CloudFormation snippet defines a maintenance window to patch on a schedule as well as an association to patch on startup:

```
MaintenanceWindow:
  Type: 'AWS::SSM::MaintenanceWindow'
  Properties:
    AllowUnassociatedTargets: false
    Duration: 2                                    ①
    Cutoff: 1                                      ②
    Name: !Ref 'AWS::StackName'
    Schedule: 'cron(0 5 ? * SUN *)'                ③
    ScheduleTimezone: UTC
MaintenanceWindowTarget:                           ④
  Type: 'AWS::SSM::MaintenanceWindowTarget'
  Properties:
    ResourceType: INSTANCE
    Targets:
    - Key: InstanceIds
      Values:
      - !Ref Instance
    WindowId: !Ref MaintenanceWindow
MaintenanceWindowTask:
  Type: 'AWS::SSM::MaintenanceWindowTask'
  Properties:
    MaxConcurrency: '1'
    MaxErrors: '1'
    Priority: 0
    Targets:
    - Key: WindowTargetIds
      Values:
      - !Ref MaintenanceWindowTarget
    TaskArn: 'AWS-RunPatchBaseline'                ⑤
    TaskInvocationParameters:
      MaintenanceWindowRunCommandParameters:
        Parameters:                                ⑥
          Operation:
```

```
            - Install
      TaskType: 'RUN_COMMAND'
      WindowId: !Ref MaintenanceWindow
  AssociationRunPatchBaselineInstall:
    Type: 'AWS::SSM::Association'                    ⑦
    Properties:
      Name: 'AWS-RunPatchBaseline'
      Parameters:
        Operation:
        - Install
      Targets:
      - Key: InstanceIds
        Values:
        - !Ref Instance
```

① The maintenance window is two hours long. You can patch more than one EC2 instance if you wish.

② The last hour is reserved for commands to finish (all commands are started in the first hour).

③ The maintenance window is scheduled every Sunday morning at 5am UTC time. Learn more about the syntax at **http://mng.bz/zmRZ**.

④ Assigns one EC2 instance to the maintenance window. You can also assign EC2 instances based on tags.

⑤ The AWS-RunPatchBaseline document is executed.

⑥ The document supports parameters. Operation can be set to Install or Scan. By default, a reboot happens if required by any patch.

⑦ The association ensures that patches are installed on startup. The same document with the same parameters are used.

There is one prerequisite missing: the EC2 instance needs read access to a set of S3 buckets for Patch Manager to work, which is granted in the next snippet. Learn more at **https://mng.bz/0ynz**:

```
InstanceRole:
  Type: 'AWS::IAM::Role'
  Properties:
    #[...]
    Policies:
    - PolicyName: PatchManager
      PolicyDocument:
        Version: '2012-10-17'
```

```
          Statement:
          - Effect: Allow
            Action: 's3:GetObject'
            Resource:
            - !Sub 'arn:aws:s3:::patch-baseline-snapshot-${AWS::Region}/
            - !Sub 'arn:aws:s3:::aws-ssm-${AWS::Region}/*'
```

Patch Manager can also visualize the patches that are waiting for installation. To gather the data, another association is needed, as shown next:

```
AssociationRunPatchBaselineScan:
  Type: 'AWS::SSM::Association'
  Properties:
    ApplyOnlyAtCronInterval: true                ①
    Name: 'AWS-RunPatchBaseline'                 ②
    Parameters:
      Operation:
      - Scan                                     ③
    ScheduleExpression: 'cron(0 0/1 * * ? *)'    ④
    Targets:
    - Key: InstanceIds
      Values:
      - !Ref Instance
```

① Do not run on startup. Unfortunately, the document AWS-RunPatchBaseline crashes when running more than once at the same time. It avoids a conflict with the association defined in AssociationRunPatchBaselineInstall.

② Uses the same document AWS-RunPatchBaseline...

③ ...but this time, Operation is set to Scan.

④ Runs every hour

It's time for a demo. Create the CloudFormation stack with the template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/ec2-os-update.yaml** by clicking the CloudFormation Quick-Create Link (**http://mng.bz/KlXn**). Pick the default Virtual Private Cloud (VPC) and subnet, then wait for the stack creation to finish.

Visit the AWS Systems Manager management console at **https://console.aws.amazon.com/systems-manager/**. Open Patch Manager in the navigation bar, and you see a nice dashboard, as shown in figure 5.3.

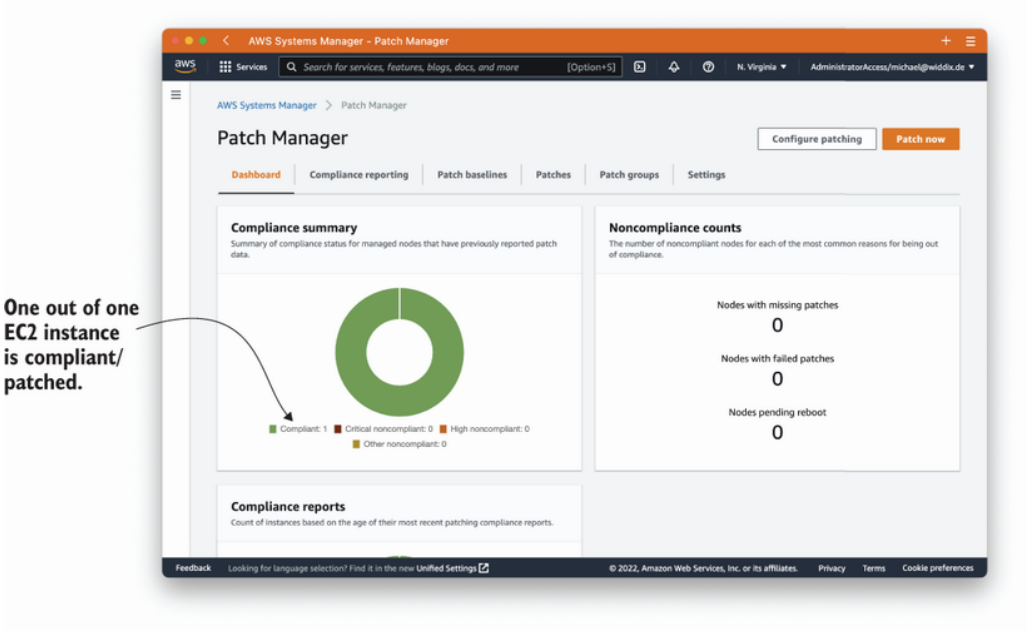

Figure 5.3 AWS Systems Manager Patch Manager dashboard

You can also patch instances manually by pressing the Patch Now button, if needed.

Cleaning up

Don't forget to delete your stack `ec2-os-update` after you finish this section, to clean up all used resources. Otherwise, you'll likely be charged for the resources you use.

5.3 Securing your AWS account

Securing your AWS account is critical. If someone gets access to your AWS account, they can steal your data, use resources at your expense, or delete all your data. As figure 5.4 shows, an AWS account is a basket for all the resources you own: EC2 instances, CloudFormation stacks, IAM users, and so on. Each AWS account comes with a root user granted unrestricted access to all resources. So far, you've used the AWS account root user to log in to the Management Console and the user `mycli`—created in section 4.2—when using the CLI. In this section, you will create an additional user to log in to the Management Console to avoid using the AWS account root user at all. Doing so allows you to manage multiple users, each restricted to the resources that are necessary for their roles.

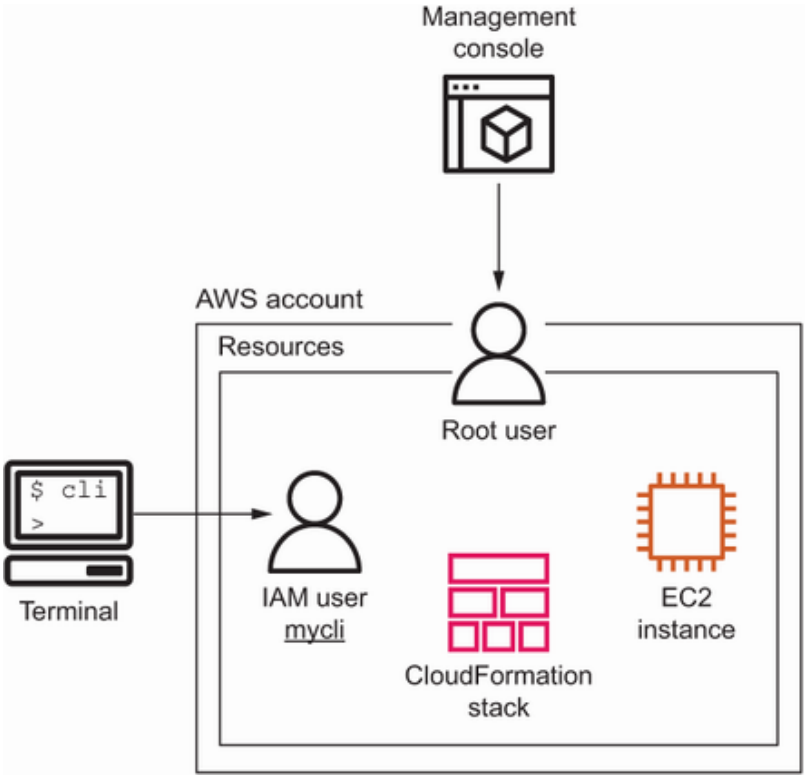

Figure 5.4 An AWS account contains all the AWS resources and comes with an AWS account root user by default.

To access your AWS account, an attacker must be able to authenticate to your account. There are three ways to do so: using the AWS account root user, using an IAM user, or authenticating as an AWS resource like an EC2 instance. To authenticate as a AWS account root user or IAM user, the attacker needs the username and password or the access keys. To authenticate as an AWS resource like an EC2 instance, the attacker needs access to the machine to communicate with the instance metadata service (IMDS).

To protect yourself from an attacker stealing or cracking your passwords or access keys, in the following section, you will enable multifactor authentication (MFA) for your AWS account root user to add an additional layer of security to the authentication process.

5.3.1 Securing your AWS account's root user

We advise you to enable MFA for the AWS account root user of your AWS account. After MFA is activated, you'll need a password and a temporary token to log in as the root user. Follow these steps to enable MFA, as shown in figure 5.5:

1. Click your name in the navigation bar at the top right of the Management Console.
2. Select Security Credentials.
3. Install an MFA app on your smartphone that supports the TOTP standard (such as Google Authenticator).

4. Expand the Multi-Factor Authentication (MFA) section.
5. Click Activate MFA.
6. Select Virtual MFA Device, and proceed with the next step.
7. Follow the instructions. Use the MFA app on your smartphone to scan the QR code that is displayed.

Figure 5.5 Protect your AWS account root user with multifactor authentication (MFA).

If you're using your smartphone as a virtual MFA device, it's a good idea not to log in to the Management Console from your smartphone or to store the AWS account root user's password on the phone. Keep the MFA token separate from your password. YubiKeys and hardware MFA tokens are also supported.

5.3.2 AWS Identity and Access Management (IAM)

Figure 5.6 shows an overview of all the core concepts of the *Identity and Access Management* (IAM) service. This service provides authentication and authorization for the AWS API. When you send a request to the AWS API, IAM verifies your identity and checks whether you are allowed to perform the action. IAM controls who (authentication) can do what (authorization) in your AWS account. For example, is the user allowed to launch a new virtual machine? The various components of IAM follow:

- An *IAM user* is used to authenticate people or workloads running outside of AWS.
- An *IAM group* is a collection of IAM users with the same permissions.
- An *IAM role* is used to authenticate AWS resources, for example, an EC2 instance.
- An *IAM identity policy* is used to define the permissions for a user, group, or role.

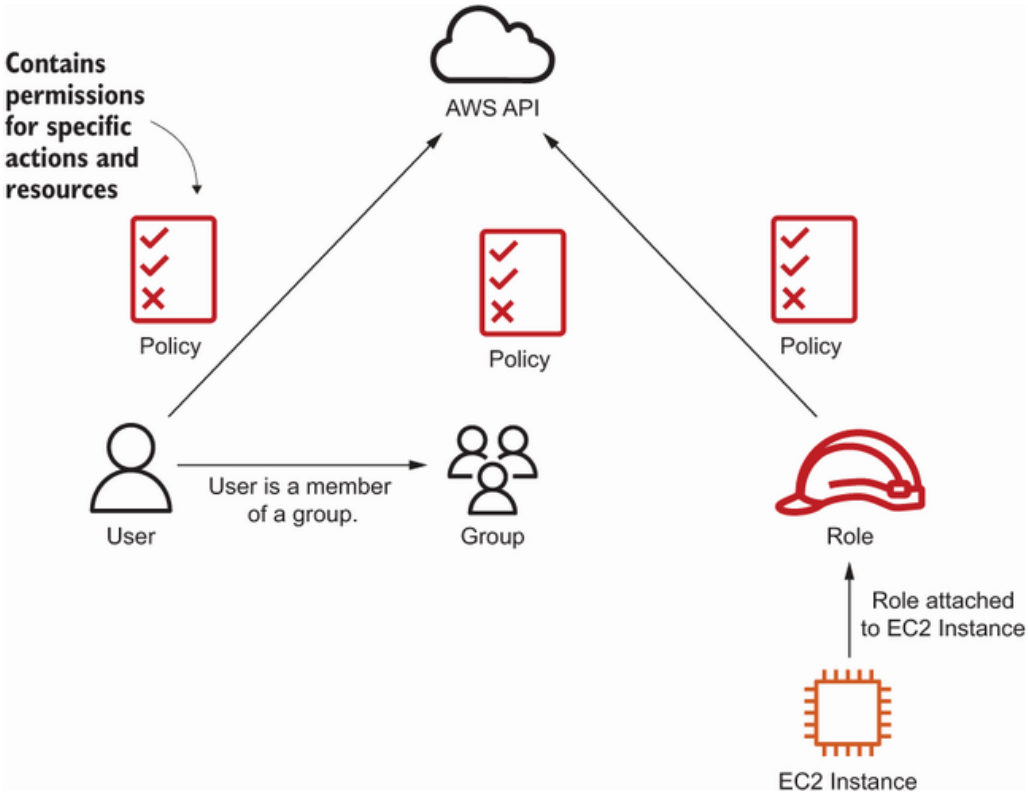

Figure 5.6 IAM concepts

Table 5.1 shows the differences between users and roles. Roles authenticate AWS entities such as EC2 instances. IAM users authenticate the people who manage AWS resources, for example, system administrators, DevOps engineers, or software developers.

Table 5.1 Differences among an AWS account root user, IAM user, and IAM role

	AWS account root user	IAM user	IAM role
Can have a password (needed to log in to the AWS Management Console)	Always	Yes	No
Can have access keys (needed to send requests to the AWS API (e.g., for CLI or SDK)	Yes (not recommended)	Yes	No
Can belong to a group	No	Yes	No
Can be associated with an EC2 instance, ECS container, Lambda function	No	No	Yes

By default, users and roles can't do anything. You have to create an identity policy stating what actions they're allowed to perform. IAM users and IAM roles use identity policies for authorization. Let's look at identity policies next.

5.3.3 Defining permissions with an IAM identity policy

By attaching one or multiple IAM identity policies to an IAM user or role, you are granting permissions to manage AWS resources. Identity policies are defined in JSON and contain one or more statements. A statement can either allow or deny specific actions on specific resources. You can use the wildcard character `*` to create more generic statements.

Identity vs. resource policies

IAM policies come in two types. *Identity policies* are attached to users, groups, or roles. *Resource policies* are attached to resources. Very few resource types support resource policies. One common example is the S3 bucket policy attached to S3 buckets.

If a policy contains the property `Principal`, it is a resource policy. The `Principal` defines who is allowed to perform the action. Keep in mind

that the principal can be set to public.
The following identity policy has one statement that allows every action for the EC2 service, for all resources:

```
{
  "Version": "2012-10-17",    ①
  "Statement": [{
    "Effect": "Allow",        ②
    "Action": "ec2:*",        ③
    "Resource": "*"           ④
  }]
}
```

① Specifies 2012-10-17 to lock down the version

② This statement allows access to actions and resources.

③ Any action offered by the EC2 service (wildcard *)...

④ ...on any resource

If you have multiple statements that apply to the same action, `Deny` overrides `Allow`. The following identity policy allows all EC2 actions except terminating EC2 instances:

```
{
   "Version": "2012-10-17",
   "Statement": [{
     "Effect": "Allow",
     "Action": "ec2:*",
     "Resource": "*"
   }, {
     "Effect": "Deny",                        ①
     "Action": "ec2:TerminateInstances",      ②
     "Resource": "*"
   }]
}
```

① Action is denied.

② Terminate EC2 instances.

The following identity policy denies all EC2 actions. The `ec2:TerminateInstances` statement isn't crucial, because `Deny` overrides `Allow`. When you deny an action, you can't allow that action with another statement:

```
{
   "Version": "2012-10-17",
   "Statement": [{
     "Effect": "Deny",
     "Action": "ec2:*",                          ①
     "Resource": "*"
   }, {
     "Effect": "Allow",
     "Action": "ec2:TerminateInstances",         ②
     "Resource": "*"
   }]
}
```

① Denies every EC2 action

② Allow isn't crucial; Deny overrides Allow.

So far, the `Resource` part has been set to `*` to apply to every resource. Resources in AWS have an Amazon Resource Name (ARN); figure 5.7 shows the ARN of an EC2 instance.

Figure 5.7 Components of an Amazon Resource Name (ARN) identifying an EC2 instance

To find out the account ID, you can use the CLI as follows:

```
$ aws sts get-caller-identity --query "Account" --output text
111111111111       ①
```

① Account ID always has 12 digits.

If you know your account ID, you can use ARNs to allow access to specific resources of a service like this:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "ec2:TerminateInstances",
    "Resource":
➥ "arn:aws:ec2:us-east-1:111111111111:instance/i-0b5c991e026104db9"
  }]
}
```

The list of all IAM actions of a service and possible resource ARNs can be found at **http://mng.bz/Po0g**.

The following two types of identity policies exist:

- *Managed policy*—If you want to create identity policies that can be reused in your account, a managed policy is what you're looking for. There are two types of managed policies:
  - *AWS managed policy*—An identity policy maintained by AWS. There are identity policies that grant admin rights, read-only rights, and so on.
  - *Customer managed*—An identity policy maintained by you. It could be an identity policy that represents the roles in your organization, for example.
- *Inline policy*—An identity policy that belongs to a certain IAM role, user, or group. An inline identity policy can't exist without the IAM role, user, or group that it belongs to.

With CloudFormation, it's easy to maintain inline identity policies; that's why we use inline identity policies most of the time in this book. One exception is the `mycli` user: this user has the AWS managed policy `AdministratorAccess` attached.

WARNING Using managed policies can often conflict with following the least-privilege principal. Managed policies usually set the `Resource` property to `*`. That's why we attach our own inline policies to IAM roles or users.

5.3.4 Users for authentication and groups to organize users

A user can authenticate using either a username and password or access keys. When you log in to the Management Console, you're authenticating with your username and password. When you use the CLI from your computer, you use access keys to authenticate as the `mycli` user.

You're using the AWS account root user at the moment to log in to the Management Console. You should create an IAM user instead, for the following reasons:

- Creating IAM users allows you to set up a unique user for every person who needs to access your AWS account.
- You can grant access only to the resources each user needs, allowing you to follow the least-privilege principle.

To make things easier if you want to add users in the future, you'll first create a group for all users with administrator access. Groups can't be used to authenticate, but they centralize authorization. So, if you want to stop your admin users from terminating EC2 instances, you need to change the identity policy only for the group instead of changing it for all admin users. A user can be a member of zero, one, or multiple groups.

It's easy to create groups and users with the CLI, as shown here. Replace `$Password` in the following with a secure password:

```
$ aws iam create-group --group-name "admin"
$ aws iam attach-group-policy --group-name "admin" \
➥ --policy-arn "arn:aws:iam::aws:policy/AdministratorAccess"
$ aws iam create-user --user-name "myuser"
$ aws iam add-user-to-group --group-name "admin" --user-name "myuser"
$ aws iam create-login-profile --user-name "myuser" --password '$Passwo
```

The user `myuser` is ready to be used. But you must use a different URL to access the Management Console if you aren't using the AWS account root user: https:// /$accountId .signin.aws.amazon.com/console. Replace `$ac-countId` with the account ID that you extracted earlier with the `aws sts get-caller-identity` command.

Enabling MFA for IAM users

We encourage you to enable MFA for all users. To enable MFA for your users, follow these steps:

1. Open the IAM service in the Management Console.
2. Choose Users at the left.
3. Click the myuser user.
4. Select the Security Credentials tab.
5. Click the Manage link near the Assigned MFA Device.
6. The wizard to enable MFA for the IAM user is the same one you used for enabling MFA for the AWS account root user.

We recommend enabling MFA for all users, especially for users granted administrator access to all or some services.

WARNING Stop using the AWS account root user from now on. Always use `myuser` and the new link to the Management Console.

WARNING You should never copy a user's access keys to an EC2 instance; use IAM roles instead! Don't store security credentials in your source

code. And never ever check them into your source code repository. Try to use IAM roles instead whenever possible, as described in the next section.

5.3.5 Authenticating AWS resources with roles

Various use cases exist where an EC2 instance needs to access or manage AWS resources. For example, an EC2 instance might need to do the following:

- Back up data to the object store S3
- Terminate itself after a job has been completed
- Change the configuration of the private network environment in the cloud

To be able to access the AWS API, an EC2 instance needs to authenticate itself. You could create an IAM user with access keys and store the access keys on an EC2 instance for authentication. But doing so is a hassle and violates security best practices, especially if you want to rotate the access keys regularly.

Instead of using an IAM user for authentication, you should use an IAM role whenever you need to authenticate AWS resources like EC2 instances. When using an IAM role, your access keys are injected into your EC2 instance automatically.

If an IAM role is attached to an EC2 instance, all identity policies attached to those roles are evaluated to determine whether the request is allowed. By default, no role is attached to an EC2 instance, and, therefore, the EC2 instance is not allowed to make any calls to the AWS API.

The following example will show you how to use an IAM role for an EC2 instance. Do you remember the temporary EC2 instances from chapter 4? What if we forgot to terminate those VMs? A lot of money would have been wasted because of that. You'll now create an EC2 instance that stops itself automatically. The following snippet shows a one-liner terminating an EC2 instance after five minutes. The command `at` is used to execute the `aws ec2 stop-instances` with a five-minute delay:

```
$ echo "aws ec2 stop-instances --instance-ids i-0b5c991e026104db9" \
➥ | at now + 5 minutes
```

The EC2 instance needs permission to stop itself. Therefore, you need to attach an IAM role to the EC2 instance. The role contains an inline identity policy granting access to the `ec2:StopInstances` action.

Unfortunately, we can't lock down the action to the EC2 instance resource itself due to a cyclic dependency. Luckily, we can grant permissions with additional conditions. One such condition is that a specific tag must be present. The following code shows how you define an IAM role with the help of CloudFormation:

```
Role:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:                    ①
      Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Principal:                               ②
          Service: 'ec2.amazonaws.com'           ③
        Action: 'sts:AssumeRole'                 ④
    ManagedPolicyArns:
    - 'arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore'
    Policies:                                    ⑤
    - PolicyName: ec2                            ⑥
      PolicyDocument:                            ⑦
        Version: '2012-10-17'
        Statement:
        - Effect: Allow                          ⑧
          Action: 'ec2:StopInstances'            ⑨
          Resource: '*'                          ⑩
          Condition:                             ⑪
            StringEquals:
              'ec2:ResourceTag/aws:cloudformation:stack-id':
➥ !Ref 'AWS::StackId'
```

① Who is allowed to assume this role?

② Specifies the principal that is allowed access to the role

③ Enter the EC2 service as the principal.

④ Allows the principal to assume the IAM role

⑤ Defines inline policies granting the role access to certain AWS resources and actions

⑥ Defines the name for the inline policy

⑦ The policy document for the inline policy

⑧ Allows...

⑨ ...stopping virtual machines...

⑩ ...for all EC2 instances...

⑪ ...but only those who are tagged with the name of the CloudFormation stack

To attach an inline role to an instance, you must first create an instance profile, as shown in the following code snippet:

```
InstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'
  Properties:
    Roles:
    - !Ref Role
```

The next code snippet shows how to attach the IAM role to the virtual machine:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
     # [...]
    IamInstanceProfile: !Ref InstanceProfile
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -ex
        TOKEN=`curl -X PUT "http://169.254.169.254/latest/api/token" \
➥ -H "X-aws-ec2-metadata-token-ttl-seconds: 21600"`
        INSTANCEID=`curl -H "X-aws-ec2-metadata-token: $TOKEN" \
➥ -s "http://169.254.169.254/latest/meta-data/instance-id"`
        echo "aws ec2 stop-instances --region ${AWS::Region} \
➥ --instance-ids $INSTANCEID" | at now + ${Lifetime} minutes
```

Create the CloudFormation stack with the template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/ec2-iam-role.yaml** by clicking the CloudFormation Quick-Create Link (**http://mng.bz/JVeP**). Specify the lifetime of the EC2 instance via the parameter, and pick the default VPC and subnet as well. Wait until the amount of time specified as the lifetime has passed, and see whether your EC2 instance is stopped in the EC2 Management Console. The lifetime begins when the server is fully started and booted.

Cleaning up
Don't forget to delete your stack `ec2-iam-role` after you finish this section, to clean up all used resources. Otherwise, you'll likely be charged for the resources you use (even when your EC2 instance is stopped, you pay for the network-attached storage).

You have learned how to use IAM users to authenticate people and IAM roles to authenticate EC2 instances or other AWS resources. You've also seen how to grant access to specific actions and resources by using an IAM identity policy. The next section will cover controlling network traffic to and from your virtual machine.

5.4 Controlling network traffic to and from your virtual machine

You want traffic to enter or leave your EC2 instance only if it has to do so. With a firewall, you control ingoing (also called *inbound* or *ingress*) and outgoing (also called *outbound* or *egress*) traffic. If you run a web server, the only ports you need to open to the outside world are ports 80 for HTTP traffic and 443 for HTTPS traffic. All other ports should be closed down. You should only open ports that must be accessible, just as you grant only the permissions you need with IAM. If you are using a firewall that allows only legitimate traffic, you close a lot of possible security holes. You can also prevent yourself from human failure—for example, you prevent accidentally sending email to customers from a test system by not opening outgoing SMTP connections for test systems.

Before network traffic enters or leaves your EC2 instance, it goes through a firewall provided by AWS. The firewalls inspects the network traffic and uses rules to decide whether the traffic is allowed or denied.

IP vs. IP address

The abbreviation IP is used for Internet Protocol, whereas an IP address describes a specific address like 84.186.116.47.

Figure 5.8 shows how an SSH request from a source IP address 10.0.0.10 is inspected by the firewall and received by the destination IP address 10.10.0.20. In this case, the firewall allows the request because a rule is in place that allows TCP traffic on port 22 between the source and the destination.

Figure 5.8 How an SSH request travels from source to destination, controlled by a firewall

AWS is responsible for the firewall, but you're responsible for the rules. By default, a security group does not allow any inbound traffic. You must add your own rules to allow specific incoming traffic. A security group contains a rule allowing all outbound traffic by default. If your use case requires a high level of network security, you should remove the rule and add your own rules to control outgoing traffic.

Debugging or monitoring network traffic

Imagine the following problem: your EC2 instance does not accept SSH traffic as you want it to, but you can't spot any misconfiguration in your firewall rules. The following two strategies are helpful:

- Use the VPC Reachability Analyzer to simulate the traffic and see if the tool finds the configuration problem. Learn more at **http://mng.bz/wyqW**.
- Enable VPC Flow Logs to get access to aggregated log messages containing rejected connections. Learn more at **http://mng.bz/qoqE**.

5.4.1 Controlling traffic to virtual machines with security groups

A security group acts as a firewall for virtual machines and other services. You will associate a security group with AWS resources, such as EC2 instances, to control traffic. It's common for EC2 instances to have more than one security group associated with them and for the same security group to be associated with multiple EC2 instances.

A security group consists of a set of rules. Each rule allows network traffic based on the following:

- Direction (inbound or outbound)
- IP protocol (TCP, UDP, ICMP)
- Port
- Source/destination based on IP address, IP address range, or security group (works only within AWS)

In theory, you could define rules that allow all traffic to enter and leave your virtual machine; AWS won't prevent you from doing so. But it's a best practice to define your rules so they are as restrictive as possible.

Security group resources in CloudFormation are of type `AWS::EC2::SecurityGroup`. The following listing is in /chapter05/firewall1.yaml in the book's code folder; the template describes an empty security group associated with a single EC2 instance.

Listing 5.1 CloudFormation template: Security group

```
---
[..]
Parameters:
  VPC:                                    ①
    # [..]
  Subnet:                                 ①
    # [..]
Resources:
  SecurityGroup:                          ②
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Learn how to protect your EC2 Instance.'
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: 'AWS in Action: chapter 5 (firewall)'
  Instance:                               ③
    Type: 'AWS::EC2::Instance'
    Properties:
      # [...]
      SecurityGroupIds:
      - !Ref SecurityGroup                ④
      SubnetId: !Ref Subnet
```

① You'll learn about this in section 5.5.

② Defines the security group without any rules (by default, inbound traffic is denied and outbound traffic is allowed). Rules will be added in the following sections.

③ Defines the EC2 instance

④ Associates the security group with the EC2 instance

To explore security groups, you can try the CloudFormation template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/firewall1.yaml**. Create a stack based on that template by clicking the CloudFormation Quick-Create Link (**http://mng.bz/91e8**), and then copy the `PublicIpAddress` from the stack output.

5.4.2 Allowing ICMP traffic

If you want to ping an EC2 instance from your computer, you must allow inbound Internet Control Message Protocol (ICMP) traffic. By default, all inbound traffic is blocked. Try `ping $PublicIpAddress` to make sure `ping` isn't working, like this:

```
$ ping 34.205.166.12
PING 34.205.166.12 (34.205.166.12): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
[...]
```

You need to add a rule to the security group that allows inbound traffic, where the protocol equals ICMP. The following listing is in /chapter05/firewall2.yaml in the book's code folder.

Listing 5.2 CloudFormation template: Security group that allows ICMP

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Learn how to protect your EC2 Instance.'
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'AWS in Action: chapter 5 (firewall)'
    SecurityGroupIngress:                                    ①
    - Description: 'allowing inbound ICMP traffic'
      IpProtocol: icmp                                       ②
      FromPort: '-1'                                         ③
      ToPort: '-1'
      CidrIp: '0.0.0.0/0'                                    ④
```

① Rules allowing incoming traffic

② Specifies ICMP as the protocol

③ ICMP does not use ports. -1 means every port.

④ Allows traffic from any source IP address

Update the CloudFormation stack with the template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/firewall2.yaml** and retry the `ping` command. It should work now:

```
$ ping 34.205.166.12
PING 34.205.166.12 (34.205.166.12): 56 data bytes
64 bytes from 34.205.166.12: icmp_seq=0 ttl=234 time=109.095 ms
64 bytes from 34.205.166.12: icmp_seq=1 ttl=234 time=107.000 ms
[...]
round-trip min/avg/max/stddev = 107.000/108.917/110.657/1.498 ms
```

Everyone's inbound ICMP traffic (every source IP address) is now allowed to reach your EC2 instance.

5.4.3 Allowing HTTP traffic

Once you can ping your EC2 instance, you want to run a web server. To do so, you must create a rule to allow inbound TCP requests on port 80, as shown in the next listing. You also need a running web server.

Listing 5.3 CloudFormation template: Security group that allows HTTP

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Learn how to protect your EC2 Instance.'
    VpcId: !Ref VPC
    # [...]
    SecurityGroupIngress:                ①
    # [...]
    - Description: 'allowing inbound HTTP traffic'
      IpProtocol: tcp                    ②
      FromPort: '80'                     ③
      ToPort: '80'                       ④
      CidrIp: '0.0.0.0/0'                ⑤
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    UserData:
      'Fn::Base64': |
        #!/bin/bash -ex
        yum -y install httpd             ⑥
        systemctl start httpd            ⑦
        echo '<html>...</html>' > /var/www/html/index.html
```

① Adds a rule to allow incoming HTTP traffic

② HTTP is based on the TCP protocol.

③ The default HTTP port is 80.

④ You can allow a range of ports or set FromPort = ToPort.

⑤ Allows traffic from any source IP address

⑥ Installs Apache HTTP Server on startup

⑦ Starts Apache HTTP Server

Update the CloudFormation stack with the template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/firewall3.yaml**. Enter the public IP address in your browser to see a very basic test page.

5.4.4 Allowing HTTP traffic from a specific source IP address

So far, you're allowing inbound traffic on port 80 (HTTP) from every source IP address. It is possible to restrict access to only your own IP address for additional security as well.

What's the difference between public and private IP addresses?

On my local network, I'm using private IP addresses that start with 192.168.0.*. My laptop uses 192.168.0.10, and my iPad uses 192.168.0.20. But if I access the internet, I have the same public IP address (such as 79.241.98.155) for my laptop and iPad. That's because only my internet gateway (the box that connects to the internet) has a public IP address, and all requests are redirected by the gateway. (If you want to know more about this, search for network address translation.) Your local network doesn't know about this public IP address. My laptop and iPad only know that the internet gateway is reachable under 192.168.0.1 on the private network.

To find your public IP address, visit **https://checkip.amazonaws.com/**. For some of us, our public IP address changes from time to time, usually when you reconnect to the internet (which happens every 24 hours in my case).

Hardcoding the public IP address into the template isn't a good solution because your public IP address can change from time to time. You already know the solution: parameters. You need to add a parameter that holds your current public IP address, and you need to modify the `Security Group`. You can find the following listing in /chapter05/firewall4.yaml in the book's code folder.

Listing 5.4 Security group allows traffic from source IP

```
Parameters:
  WhitelistedIpAddress:                                          ①
    Description: 'Whitelisted IP address'
    Type: String
    AllowedPattern: '^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$'
    ConstraintDescription: 'Enter a valid IPv4 address'
Resources:
  SecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Learn how to protect your EC2 Instance.'
      VpcId: !Ref VPC
      # [...]
      SecurityGroupIngress:
      # [...]
      - Description: 'allowing inbound HTTP traffic'
        IpProtocol: tcp
        FromPort: '80'
        ToPort: '80'
        CidrIp: !Sub '${WhitelistedIpAddress}/32'                ②
```

① Public IP address parameter

② Uses WhitelistedIpAddress/32 as a value to turn the IP address input into a CIDR

Update the CloudFormation stack with the template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/firewall4.yaml**. When asked for parameters, type in your public IP address for `WhitelistedIpAddress`. Now only your IP address can open HTTP connections to your EC2 instance.

Classless Inter-Domain Routing (CIDR)

You may wonder what `/32` means. To understand what's going on, you need to switch your brain into binary mode. An IP address is 4 bytes or 32 bits long. The `/32` defines how many bits (32, in this case) should be used to form a range of addresses. If you want to define the exact IP address that is allowed, you must use all 32 bits.

Sometimes it makes sense to define a range of allowed IP addresses. For example, you can use `10.0.0.0/8` to create a range between 10.0.0.0 and 10.255.255.255, `10.0.0.0/16` to create a range between 10.0.0.0 and 10.0.255.255, or `10.0.0.0/24` to create a range between 10.0.0.0 and 10.0.0.255. You aren't required to use the binary boundaries (8, 16, 24, 32),

but they're easier for most people to understand. You already used `0.0.0.0/0` to create a range that contains every possible IP address. Now you can control network traffic that comes from outside a virtual machine or goes outside a virtual machine by filtering based on protocol, port, and source IP address.

5.4.5 Allowing HTTP traffic from a source security group

It is possible to control network traffic based on whether the source or destination belongs to a specific security group. For example, you can say that a MySQL database can be accessed only if the traffic comes from your web servers, or that only your proxy servers are allowed to access the web servers. Because of the elastic nature of the cloud, you'll likely deal with a dynamic number of virtual machines, so rules based on source IP addresses are difficult to maintain. This process becomes easy, however, if your rules are based on source security groups.

To explore the power of rules based on a source security group, let's look at the concept of a load balancer/ingress router/proxy. The client sends requests to the proxy. The proxy inspects the request and forwards it to the backend. The backend response is then passed back to the client by the proxy. To implement the concept of an HTTP proxy, you must follow these two rules:

- Allow HTTP access to the proxy from 0.0.0.0/0 or a specific source address.
- Allow HTTP access to all backends only if the traffic source is the proxy.

Figure 5.9 shows that only the proxy is allowed to communicate with the backend over HTTP.

Figure 5.9 The proxy is the only HTTP entry point to the system (realized with security groups).

A security group allowing incoming HTTP traffic from anywhere needs to be attached to the proxy. All backend VMs are attached to a security group allowing HTTP traffic only if the source is the proxy's security group. Listing 5.5 shows the security groups defined in a CloudFormation template.

Listing 5.5 CloudFormation template: HTTP from proxy to backend

```
SecurityGroupProxy:                                          ①
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Allowing incoming HTTP and ICPM from anywhere.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - Description: 'allowing inbound ICMP traffic'
      IpProtocol: icmp
      FromPort: '-1'
      ToPort: '-1'
      CidrIp: '0.0.0.0/0'
    - Description: 'allowing inbound HTTP traffic'
      IpProtocol: tcp
      FromPort: '80'
      ToPort: '80'
      CidrIp: '0.0.0.0/0'
SecurityGroupBackend:                                        ②
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Allowing incoming HTTP from proxy.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - Description: 'allowing inbound HTTP traffic from proxy'
      IpProtocol: tcp
      FromPort: '80'
      ToPort: '80'
      SourceSecurityGroupId: !Ref SecurityGroupProxy         ③
```

① Security group attached to the proxy

② Security group attached to the backend

③ Allows incoming HTTP traffic only from proxy

Update the CloudFormation stack with the template located at **https://s3.amazonaws.com/awsinaction-code3/chapter05/firewall5.yaml**. After the update is completed, the stack will show the following two outputs:

- `ProxyPublicIpAddress`—Entry point into the system. Receives HTTP requests from the outside world.
- `BackendPublicIpAddress`—You can connect to this EC2 instance only from the proxy.

Copy the `ProxyPublicIpAddress` output and open it in your browser. A Hello AWS in Action! page shows up.

But when you copy the `BackendPublicIpAddress` output and try to open it in your browser, an error message will appear. That's because the security group of the backend instance allows incoming traffic only from the proxy instance, not from your IP address.

The only way to reach the backend instance is through the proxy instance. The Hello AWS in Action! page that appears when opening http://$ProxyPublicIpAddress in your browser originates from the backend instance but gets passed through the proxy instance. Check the details of the HTTP request, for example, with `curl`, as shown in the following code. You'll find a `x-backend` response header indicating that the proxy forwarded your request to the backend named `app1`, which points to the backend instance:

```
$ curl -I http://$ProxyPublicIpAddress
< HTTP/1.1 200 OK
[...]
< accept-ranges: bytes
< content-length: 91
< content-type: text/html; charset=UTF-8
< x-backend: app1                                    ①
<html><title>Hello AWS in Action!</title><body>
➥ <h1>Hello AWS in Action!</h1></body></html>
```

① The x-backend header is injected by the proxy and indicates the backend answered the request.

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

5.5 Creating a private network in the cloud: Amazon Virtual Private Cloud (VPC)

When you create a VPC, you get your own private network on AWS. *Private* means you can use the address ranges 10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16 to design a network that isn't necessarily connected to the public internet. You can create subnets, route tables, network access control lists (NACLs), and gateways to the internet or a VPN endpoint.

IPv6

Amazon Virtual Private Cloud supports IPv6 as well. You can create IPv4 only, IPv6 only, or IPv4 and IPv6 VPCs. To reduce complexity, we are sticking to IPv4 in this chapter.

Subnets allow you to separate concerns. We recommend to create at least the following two types of subnets:

- *Public subnets*—For all resources that need to be reachable from the internet, such as a load balancer of a internet-facing web application
- *Private subnets*—For all resources that should not be reachable from the internet, such as an application server or a database system

What's the difference between a public and private subnet? A public subnet has a route to the internet; a private subnet doesn't.

For the purpose of understanding how a VPC works, you'll create a VPC to replicate the example from the previous section. You'll implement the proxy concept from the previous section by creating a public subnet that contains only the proxy. You'll also create a private subnet for the backend servers. You will not be able to access a backend server directly from the internet because it will sit on a private subnet. The backend servers will be accessible only via the proxy server running in a public subnet.

The VPC uses the address space 10.0.0.0/16. To isolate different parts of the system, you'll add the next two public subnets and one private subnet to the VPC:

- *10.0.0.0/24*—Public subnet used later in this section to deploy a NAT gateway
- *10.0.1.0/24*—Public proxy subnet
- *10.0.2.0/24*—Private backend subnet

What does 10.0.0.0/16 mean?

10.0.0.0/16 represents all IP addresses in 10.0.0.0 and 10.0.255.255. It's using CIDR notation (explained earlier in the chapter).

Figure 5.10 shows the architecture of the VPC.

Figure 5.10 VPC with three subnets to secure a web application

You'll use CloudFormation to describe the VPC with its subnets. The template is split into smaller parts to make it easier to read in the book. As usual, you'll find the code in the book's code repository on GitHub: **https://github.com/AWSinAction/code3**. The template is located at /chapter05/vpc.yaml.

5.5.1 Creating the VPC and an internet gateway (IGW)

The first resources listed in the template are the VPC and the internet gateway (IGW). In the next code snippet, the IGW will translate the public IP addresses of your virtual machines to their private IP addresses using network address translation (NAT). All public IP addresses used in the VPC are controlled by this IGW:

```
VPC:
  Type: 'AWS::EC2::VPC'
  Properties:
    CidrBlock: '10.0.0.0/16'                ①
    EnableDnsHostnames: 'true'
    Tags:                                   ②
    - Key: Name
      Value: 'AWS in Action: chapter 5 (VPC)'
InternetGateway:                            ③
  Type: 'AWS::EC2::InternetGateway'
  Properties: {}
VPCGatewayAttachment:                       ④
  Type: 'AWS::EC2::VPCGatewayAttachment'
  Properties:
    VpcId: !Ref VPC
    InternetGatewayId: !Ref InternetGateway
```

① The IP address space used for the private network

② Adds a Name tag to the VPC

③ An IGW is needed to enable traffic to and from the internet.

④ Attaches the internet gateway to the VPC

Next you'll define the subnet for the proxy.

5.5.2 Defining the public proxy subnet

The EC2 instance running the proxy needs to be reachable via the internet. To achieve that, you'll need to complete the next four steps:

1. Create a subnet spanning a subsection of the IP address range assigned to the VPC.
2. Create a route table, and attach it to the subnet.
3. Add the `0.0.0.0/0` route pointing to the internet gateway to the route table.
4. Create an NACL, and attach it to the subnet.

To allow traffic from the internet to the proxy machine and from the proxy to the backend servers, you'll need the following NACL rules, shown in the next code snippet:

- Internet to proxy: HTTP from 0.0.0.0/0 to 10.0.1.0/24 is allowed.
- Proxy to backend: HTTP from 10.0.1.0/24 to 10.0.2.0/24 is allowed.

```
SubnetPublicProxy:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: !Select [0, !GetAZs '']      ①
    CidrBlock: '10.0.1.0/24'                       ②
    MapPublicIpOnLaunch: true
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'Public Proxy'
RouteTablePublicProxy:                             ③
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPublicProxy:                  ④
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicProxy
    RouteTableId: !Ref RouteTablePublicProxy
RoutePublicProxyToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePublicProxy
    DestinationCidrBlock: '0.0.0.0/0'              ⑤
    GatewayId: !Ref InternetGateway
  DependsOn: VPCGatewayAttachment
NetworkAclPublicProxy:                             ⑥
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPublicProxy:            ⑦
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
```

```
      SubnetId: !Ref SubnetPublicProxy
      NetworkAclId: !Ref NetworkAclPublicProxy
```

① Picks the first availability zone in the region. (You'll learn about availability zones in chapter 16.)

② IP address space

③ Route table

④ Associates the route table with the subnet

⑤ Routes everything (0.0.0.0/0) to the IGW

⑥ Network access control list (NACL)

⑦ Associates the NACL with the subnet

There's an important difference between security groups and NACLs: security groups are stateful, but NACLs aren't. If you allow an inbound port on a security group, the corresponding response to requests on that port are allowed as well. A security group rule will work as you expect it to. If you open inbound port 80 on a security group, you can connect via HTTP.

That's not true for NACLs. If you open inbound port 80 on an NACL for your subnet, you still may not be able to connect via HTTP. In addition, you need to allow outbound ephemeral ports, because the web server accepts connections on port 80 but uses an ephemeral port for communication with the client. Ephemeral ports are selected from the range starting at 1024 and ending at 65535. If you want to make an HTTP connection from within your subnet, you have to open outbound port 80 and inbound ephemeral ports as well.

Another difference between security group rules and NACL rules is that you have to define the priority for NACL rules. A smaller rule number indicates a higher priority. When evaluating an NACL, the first rule that matches a package is applied; all other rules are skipped.

The proxy subnet allows clients sending HTTP requests on port 80. The following NACL rules are needed:

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

We also want to make HTTP and HTTPS requests from the subnet to the internet. The following NACL rules 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 next:

```
NetworkAclEntryInPublicProxyHTTP:                    ①
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '110'                                ②
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'false'                                  ③
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryInPublicProxyEphemeralPorts:          ④
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicProxyHTTP:                   ⑤
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '80'
      To: '80'
    RuleAction: 'allow'
    Egress: 'true'                                   ⑥
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryOutPublicProxyHTTPS:                  ⑦
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicProxy
```

```
      RuleNumber: '110'
      Protocol: '6'
      PortRange:
        From: '443'
        To: '443'
      RuleAction: 'allow'
      Egress: 'true'
      CidrBlock: '0.0.0.0/0'
  NetworkAclEntryOutPublicProxyEphemeralPorts:      ⑧
    Type: 'AWS::EC2::NetworkAclEntry'
    Properties:
      NetworkAclId: !Ref NetworkAclPublicProxy
      RuleNumber: '200'
      Protocol: '6'
      PortRange:
        From: '1024'
        To: '65535'
      RuleAction: 'allow'
      Egress: 'true'
      CidrBlock: '0.0.0.0/0'
```

① Allows inbound HTTP from everywhere

② Rules are evaluated starting with the lowest-numbered rule.

③ Inbound

④ Ephemeral ports used for short-lived TCP/IP connections

⑤ Allows outbound HTTP to everywhere

⑥ Outbound

⑦ Allows outbound HTTPS to everywhere

⑧ Ephemeral ports

We recommend you start with using security groups to control traffic. If you want to add an extra layer of security, you should use NACLs on top. But doing so is optional, in our opinion.

5.5.3 Adding the private backend subnet

As shown in figure 5.11, the only difference between a public and a private subnet is that a private subnet doesn't have a route to the IGW.

Traffic between subnets of a VPC is always routed by default. You can't remove the routes between the subnets. If you want to prevent traffic between subnets in a VPC, you need to use NACLs attached to the subnets, as shown here. The subnet for the web server has no additional routes and is, therefore, private:

```
SubnetPrivateBackend:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: !Select [0, !GetAZs '']
    CidrBlock: '10.0.2.0/24'      ①
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'Private Backend'
RouteTablePrivateBackend:         ②
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPrivateBackend:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPrivateBackend
    RouteTableId: !Ref RouteTablePrivateBackend
NetworkAclPrivateBackend:
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPrivateBackend:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetPrivateBackend
    NetworkAclId: !Ref NetworkAclPrivateBackend
```

① Address space

② No route to the IGW

Figure 5.11 Private and public subnets

The backend subnet allows HTTP requests from the proxy subnet. The following NACLs are needed:

- Allow inbound port 80 (HTTP) from 10.0.1.0/24.
- Allow outbound ephemeral ports to 10.0.1.0/24.

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:                    ①
  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:          ②
  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:                   ③
  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:                  ④
  Type: 'AWS::EC2::NetworkAclEntry'
```

```
    Properties:
      NetworkAclId: !Ref NetworkAclPrivateBackend
      RuleNumber: '110'
      Protocol: '6'
      PortRange:
        From: '443'
        To: '443'
      RuleAction: 'allow'
      Egress: 'true'
      CidrBlock: '0.0.0.0/0'
  NetworkAclEntryOutPrivateBackendEphemeralPorts:        ⑤
    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'
```

① Allows inbound HTTP from proxy subnet

② Ephemeral ports

③ Allows outbound HTTP to everywhere

④ Allows outbound HTTPS to everywhere

⑤ 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                ①
    SubnetId: !Ref SubnetPublicProxy    ②
    Tags:
    - Key: Name
      Value: Proxy
    UserData: # [...]
  DependsOn: VPCGatewayAttachment       ③
```

① This security group allows everything.

② Launches in the proxy subnet

③ 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    ①
    Tags:
    - Key: Name
      Value: Backend
    UserData:
      'Fn::Base64': |
        #!/bin/bash -ex
        yum -y install httpd                ②
        systemctl start httpd               ③
        echo '<html>...</html>' > /var/www/html/index.html
```

① Launches in the private backend subnet

② Installs Apache from the internet

③ 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'                    ①
    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:                       ②
  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
```

① 10.0.0.0/24 is the NAT subnet.

② 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:                ①
  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:               ②
  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:      ③
  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:               ④
```

```
    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:                ⑤
    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:       ⑥
    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'
```

① Allows inbound HTTP from all VPC subnets

② Allows inbound HTTPS from all VPC subnets

③ Ephemeral ports

④ Allows outbound HTTP to everywhere

⑤ Allows outbound HTTPS to everywhere

⑥ 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:                                    ①
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: 'vpc'
NatGateway:                                       ②
  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                 ③
```

① A static public IP address is used for the NAT gateway.

② The NAT gateway is placed into the private subnet and associated with the static public IP address.

③ 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        ①
```

① 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.