Assignment 1 - Mini Network

Overview  
Assignment to build a mini-network for a business based on the two figures in the problem using Cloud Formation.  
Infrastructure as Code.   
  
Student ID: L00177579  
Ryan Clarke  
Github Repository: <https://github.com/L00177579/CFAssignment1>

For IP address context on the figures below containing EC2 instances:

JumpBox:

* Public IP: 3.81.80.175
* Private IP: 10.10.3.55

FrontEnd:

* Public IP: 18.212.146.92
* Private IP: 10.10.4.26

Application:

* Public IP: None
* Private IP: 10.10.1.76

Database:

* Public IP: None
* Private IP: 10.10.1.92

For the figures on checking the IP Address:

* NAT Gateway Elastic Public IP: 3.212.192.247
* Front End Public IP: 3.93.2.66
* Front End Private IP: 10.10.4.18
* Database Private IP: 10.10.1.231

Conclusion  
The purpose of this assignment was to implement a mini network for a business based on the supplied figures. In this conclusion I will discuss the research, best practices, networking, templating, security, testing, issues, and high availability applied to the resulting network. Additionally I will briefly touch on a comparison to a competitor product with Microsofts Azure: Bicep/ARM.

To begin I created a VPC with a specified CIDR block. AWS has a fixed range for the network prefix length with 16 giving the biggest range and 28 giving the smallest. For this mini network testing I have gone with the CIDR block of 10.10.0.0/16 for the VPC. This gives us a total of 65,536 ip addresses that can be used (minus the 5 addresses reserved per each subnet). For the subnets I've used a 24 net prefix length. An example for the private subnet hosting the Database EC2 instance has a CIDR block of 10.10.2.0/24. This allows for 250 addresses to be assigned within the subnet. It was difficult to find information on the advantages and disadvantages of using the above ranges. When creating a large VPC (VPCs in AWS max at a prefix of 16) the advantage is room to expand with additional subnets and other resources that require an IP address. The only disadvantage I could find is when it comes to have a large ip address space for VPCs is around VPC peering. Two VPCs connecting to each other can't have overlapping CIDR blocks. Given the above block I set for the VPC and the use case of a network for a small business it's unlikely they would hit the 256 VPC limit for the 10.0.0.0/8 block (Assuming /16 prefixes for each VPC). Additionally there's other IP ranges as specified in the RFC 1918: 172.16.0.0/12 and 192.168.0.0/16 that can be used if necessary. To sum up I believe we're safe to go with these IP ranges for future expansion and without risk on VPC peering issues. If good practices and infrastructure as code is followed we should be able to change the resources as the need arises.

Following an Infrastructure as Code approach (specifically the benefit of scalability) I wanted to break down the initial network template into multiple re-usable pieces of yaml. The templates I ended up with are as follows: vpc, subnet, jumpbox, frontend, database, application and mysql. Each of these can then be reused by any in part or as a whole to make a network by anyone with access to the files. This is useful on a team or in an organisation where have a set of tested working templates will save time and money. Each template has it's own set of parameters and outputs that can be referenced in other templates. The VPC template is a simple template that returns an important output; a reference to the VPC Id. This is needed in most of other templates as many resources require the VPC Id as an input. For the subnet template I wanted it to work for private and public subnets. This meant public subnets needed to be able to create a NAT Gateway but use the Internet Gateway from the VPC, while private subnets needed to be able to use the a NAT Gateway reference. By using default parameter values and conditions the subnet template achieves this allowing for a configurable creation of a subnet depending on needs. For example in my main stack template, to save on costs, I've only created one NAT Gateway on the first public subnet. This then gets referenced by both private subnets. The subnet template returns relevant references required for the EC2 instances (and for the subnets in the case of the NAT Gateway). I've given each EC2 instance from the requirements their own template because each instance has it's own set of security requirements and inputs. Lastly, the mysql template contains the relevant security groups required for the mysql database that I used for testing. The idea I have here is that any additional databases such as SQL Server could be added as templates then passed into the front end and database EC2 instance template allowing them to have the required security groups. By removing all the additional setup and placing it into their own templates I've significantly reduced the size of the main template. This is a benefit that becomes more apparent as more resources are added to the main template. Additionally if any resources need to be removed there's no guesswork on what parts need to be removed, it is simply drop the part referencing the *AWS::CloudFormation::Stack*. Furthermore having each item called as a *AWS::CloudFormation::Stack* breaks up the main stack into nested stacks as in Figure 2. All in all, I believe breaking the network into several templates has tidied up the main template dramatically (especially in line count) but at a small cost of some extra complexity in the resource templates. But this extra complexity is mitigated by each piece being a shareable tested template.

For each of these templates I followed the best practices as outlined by AWS in their official documentation. I made sure to test early and frequently when making my templates so that syntax errors and configuration issues were not pushed to the repository. Every resource that could have a Name tag was given one, this is important for future work requiring a cross stack reference. Each of these name tags also used the pseudo-parameter *AWS::StackName* so that it would fulfill the requirement of each name being unique. Template parameters used AWS specific types when supported (some are parameters use string as the parameter type was not supported) and constraints when they required a specific set of values. Files were stored and tracked using git and pushed (via branches) to a remote repository on Github. I've also commented where I thought was necessary to clear up what parts are doing. Each of these greatly improve the consistency of the code and I'd imagine as more work would be done for this business there'd be an implementation of organisation specific best practices such as versioning.

For security there's a few parts that bring it all together into a working state. The first is private subnets are not accessible from outside the VPC. They have no public facing IP address and port 22 is not open for inbound traffic so you can't SSH into them from outside the network. This is where I've differed from the final figure in the assignment, the elastic IP in the private subnet for the application EC2 instance defeats the purpose of it being a private subnet. Instead to access resources in private subnets you need to SSH onto the jump server using a private key (this configured for the EC2 instances through the SSHKeyPair parameters) then SSH onto instances in the private or public subnet using a private key as shown in Figures 3 and 5. To allow SSH from outside the network onto the jump server I created a security group for the jump server EC2 instance. The Jump Box security group opens up the inbound port 22 for the CIDR 0.0.0.0/0 (the internet) and the outbound port 22 for the IP range of the VPC. As another layer of security this Jump Box security group is passed as a reference to other EC2 instance security groups as a Source or Destination group. This Jump Box security group is passed into each EC2 instance template to allow SSH access. In addition to the Jump Box reference, each instance has its own security group for their specific needs. For example the front end EC2 instance requires users to be able to access webpages so I've opened up the HTTP and HTTPs inbound ports 80 and 443 on the 0.0.0.0/0 CIDR range. This allows users to access the hosted webpage from outside the VPC as seen in Figure 9. The front end and database EC2 instances also have these HTTP/HTTPS ports open for their outbound traffic. This let me download and install the webserver and database software. For the front end on a public subnet its IP address is the assigned public one as shown in Figure 10 but for the database on a private subnet it will show as the NAT Gateway IP address as shown in Figure 12. Having a NAT Gateway allows resources on private instances to access the internet without having their own public IP Address (and subsequently opening them up to attack). In the case of the database we don't want it to be publicly available but we do need to download the database software.

References & Bibliography  
Perry, T. (2015) *IPv4, CIDR, and Amazon VPCs in a Nutshell.* Available at: <https://treyperry.com/2015/06/22/ipv4-cidr-vpc-in-a-nutshell/>  
(Accessed 1 November 2022).

*Amazon EC2 instance IP addressing* (n.d.). Available at: <https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/using-instance-addressing.html>  
(Accessed 1 November 2022).

*Why use infrastructure as code?* (n.d.) Available at:  
<https://containersonaws.com/introduction/infrastructure-as-code/>  
(Accessed 1 November 2022).

Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, Available at: <https://www.rfc-editor.org/info/rfc1918>  
(Accessed 2 November 2022)

*VPC peering basics* (n.d.) Available at:  
<https://docs.aws.amazon.com/vpc/latest/peering/vpc-peering-basics.html>  
(Accessed 2 November 2022)

*AWS CloudFormation best practices* (n.d.) Available at:  
<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html>  
(Accessed 3 November 2022)

Appendices  
  
Figure 1 - EC2 Instances, the top two instances have public IPV4 addresses.

  
Figure 2 - At the bottom the main stack, above that the nested stacks.

  
Figure 3 - SSH into Jump Server EC2 instance using PuTTY and private key.

  
Figure 4 - Trying to SSH onto the other public IP address on the Front End EC2 instance.

  
Figure 5 - Using scp command to copy the private key onto the Jump Box for use in connecting to other EC2 instances.

  
Figure 6 - SSH into another EC2 instance from the jump box.

  
Figure 7 - Using curl to access HTTP public resources.

  
Figure 8 - curl command doesn't complete on Jump Server due to ports 80/443 not being open.

  
Figure 9 - Front End EC2 instance Apache webserver accessing Database EC2 Instance MySQL database using port 3306.

  
Figure 10 - Using curl to return the public IP Address of the front end EC2 instance. In this case it has it's own public IP.

  
Figure 11 - Elastic IP Assigned to the NAT Gateway.

  
Figure 12 - Using curl to return the public IP Address of the database EC2 instance.  
In this case it's going through the NAT Gateway.