How do you share resources between multiple CloudFormation stacks?
Or
What are nested stacks and cross-stack references? When would you use them?

I've used both **nested stacks** and **cross-stack references** extensively to manage scalable, modular, and reusable infrastructure using AWS CloudFormation. These approaches have been crucial in keeping our infrastructure organized and maintainable, especially in larger projects.

🔷 What are Nested Stacks?

Definition: A **nested stack** is essentially a child stack embedded within a parent CloudFormation stack.
It helps break down large, complex templates into smaller, focused units.

Why I Use Them:

- To stay under the **51,200-character limit** of a template
- To **reuse infrastructure patterns** like VPC, subnets, security groups
- To allow **teams to independently own and manage** parts of the infrastructure
- To simplify **lifecycle management** (e.g., compute layer changes frequently, but the networking layer is stable)

Real Usage:
In one project, our parent template included nested stacks for:

- Networking (networking.yaml)
- Compute resources (compute.yaml)
- Monitoring setup (monitoring.yaml)

Each nested template lived in S3 and was version-controlled independently.

🔶 What are Cross-Stack References?

Definition: Cross-stack references allow a stack to **export outputs**, and other stacks can **import those values**.

Why I Use Them:

- To **share common infrastructure** like VPCs, security groups, ALBs, or RDS endpoints across multiple stacks
- To support **microservices architecture**, where each service stack is loosely coupled
- To separate responsibilities — for example, networking managed by one team, applications by another

---

🛠️ How I Share Resources Between Stacks

✅ Method 1: Cross-Stack Exports/Imports

Step 1: Export in source stack:

yaml

CopyEdit

```
Outputs:
  VpcId:
    Value: !Ref MyVPC
    Export:
      Name: 'networking-VpcId'
```

Step 2: Import in dependent stack:

yaml

CopyEdit

```
Resources:
  MyEC2Instance:
    Properties:
      VpcId: !ImportValue 'networking-VpcId'
```

✅ Method 2: SSM Parameter Store

I also store resource info like DB endpoints in SSM and reference them using {{resolve:ssm:/myapp/database}}. This makes it more dynamic and loosely coupled.

✅ Method 3: Pass as Parameters

When creating Stack B, I fetch outputs from Stack A and pass them using --parameters.

📦 Real-World Example: E-commerce Platform

We split the infra into:

- **Foundation Stack** (nested): VPC, subnets, shared ALB
- **Database Stack**: RDS, ElastiCache, exported endpoints
- **Application Stacks**: User/Product/Order services importing shared infra

Each stack was independently deployed and owned by a specific team.

⚙️ When to Use Each Approach

Use Case	Use Nested Stacks	Use Cross-Stack References
Modularity and maintainability	✅	✅
Large templates approaching limits	✅	❌
Reusable components	✅ (standard networking, monitoring)	✅ (shared VPC, SG, DB endpoint)
Microservices architecture	❌	✅
Loose coupling between teams	✅	✅

Use Case	Use Nested Stacks	Use Cross-Stack References
Lifecycle separation	✅	✅

🧠 Best Practices I Follow

For Nested Stacks:

- Single responsibility per nested stack
- Version templates in S3
- Use DependsOn to manage stack order
- Test nested stacks individually before integration

For Cross-Stack References:

- Use descriptive export names (prefix with stack name)
- Keep exports minimal — only what's needed
- Carefully plan deletion order (consumers must be deleted first)
- Avoid circular dependencies

⚠️ Common Pitfalls

- **Changing export names**: breaks downstream stacks
- **Too many cross-stack references**: leads to tight coupling
- **Circular dependencies**: difficult to untangle
- **Forgetting S3 URL for nested stacks**: they must live in S3

🚫 When NOT to Use

- Avoid nested stacks for small or simple templates
- Avoid cross-stack references when:
  - Resources change frequently together
  - You need fully atomic deployments (better handled in one stack)

✅ Conclusion

Using **nested stacks** and **cross-stack references** has helped me build clean, maintainable infrastructure across multiple environments and teams. The key is to **balance modularity with simplicity** and use these tools when they add clarity, not complexity.

NESTED STACK

1. Child Stack (e.g., webserver-stack.yaml)

This stack defines the web server resources.

YAML

```
# webserver-stack.yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: A simple web server stack

Parameters:
  VpcId:
    Type: String
    Description: The ID of the VPC where the web server will be deployed.
  SubnetId:
    Type: String
    Description: The ID of the subnet where the web server will be deployed.
  SecurityGroupId:
    Type: String
    Description: The ID of the security group to associate with the web server.

Resources:
  WebServerInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-0abcdef1234567890 # Replace with a valid AMI ID for your region
      InstanceType: t2.micro
      NetworkInterfaces:
        - DeviceIndex: "0"
          AssociatePublicIpAddress: "true"
          SubnetId: !Ref SubnetId
          GroupSet:
            - !Ref SecurityGroupId
      Tags:
        - Key: Name
          Value: MyWebServer

Outputs:
  WebServerPublicIp:
    Description: The public IP address of the web server
    Value: !GetAtt WebServerInstance.PublicIp
```

2. Parent Stack (e.g., parent-network-and-app-stack.yaml)

This stack defines the network and then deploys the webserver-stack as a nested stack.

YAML

```
# parent-network-and-app-stack.yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Parent stack for network and web application deployment

Resources:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsSupport: true
      EnableDnsHostnames: true
      Tags:
        - Key: Name
          Value: MyVPC

  PublicSubnet:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.0.1.0/24
      MapPublicIpOnLaunch: true
      Tags:
        - Key: Name
          Value: MyPublicSubnet
  WebServerSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupName: WebServerSG
      GroupDescription: Enable HTTP access
      VpcId: !Ref VPC
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 80
          ToPort: 80
          CidrIp: 0.0.0.0/0
      Tags:
        - Key: Name
          Value: WebServerSecurityGroup

  # Nested Stack for the Web Server
  WebServerStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: webserver-stack.yaml # Assuming both files are in the same S3 bucket or local path
      Parameters:
        VpcId: !Ref VPC
        SubnetId: !Ref PublicSubnet
        SecurityGroupId: !Ref WebServerSecurityGroup

Outputs:
  WebServerErrorCheck:
    Description: Public IP of the web server from the nested stack
    Value: !GetAtt WebServerStack.Outputs.WebServerPublicIp
```

Explanation:

- **webserver-stack.yaml (Child Stack):** This is a self-contained template that deploys an EC2 instance. It takes VpcId, SubnetId, and SecurityGroupId as parameters, which will be provided by the parent stack. It outputs the public IP of the web server.
- **parent-network-and-app-stack.yaml (Parent Stack):**
  - It first creates a VPC, a public subnet, and a security group.
  - The crucial part is the WebServerStack resource. Its Type is AWS::CloudFormation::Stack, which indicates it's a nested stack.
  - TemplateURL: This specifies the S3 URL or local path to the child stack template (webserver-stack.yaml).
  - Parameters: This is how the parent stack passes values to the child stack. It's mapping the !Ref (references) to its own created resources (VPC, Subnet, Security Group) to the parameters expected by the webserver-stack.yaml.
  - Outputs: The parent stack can also retrieve outputs from the nested stack using !GetAtt WebServerStack.Outputs.WebServerPublicIp.

How to deploy (Conceptual):

1. Upload webserver-stack.yaml and parent-network-and-app-stack.yaml to an S3 bucket (or deploy locally if using the AWS CLI and have the files in the same directory).

2. Deploy the parent-network-and-app-stack.yaml using the CloudFormation console or AWS CLI. CloudFormation will then automatically create the nested webserver-stack as part of its deployment.

CROSS REFRENCE STACK

1. Network Stack (network-stack.yaml) - The Exporter

This stack creates the network resources and **exports** the PublicSubnetId and VpcId.

YAML

```
# network-stack.yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Creates a VPC and a public subnet, then exports their IDs.

Resources:
  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsSupport: true
      EnableDnsHostnames: true
      Tags:
        - Key: Name
          Value: MySharedVPC

  MyPublicSubnet:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.1.0/24
      MapPublicIpOnLaunch: true
      Tags:
        - Key: Name
          Value: MySharedPublicSubnet

  MyInternetGateway:
    Type: AWS::EC2::InternetGateway
    Properties:
      Tags:
        - Key: Name
          Value: MyInternetGateway

  AttachGateway:
    Type: AWS::EC2::VPCGatewayAttachment
    Properties:
      VpcId: !Ref MyVPC
      InternetGatewayId: !Ref MyInternetGateway

  PublicRouteTable:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref MyVPC
      Tags:
        - Key: Name
          Value: MyPublicRouteTable

  PublicRoute:
    Type: AWS::EC2::Route
    DependsOn: AttachGateway # Ensure IGW is attached before creating route
    Properties:
      RouteTableId: !Ref PublicRouteTable
      DestinationCidrBlock: 0.0.0.0/0
      GatewayId: !Ref MyInternetGateway

  SubnetRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      SubnetId: !Ref MyPublicSubnet
      RouteTableId: !Ref PublicRouteTable

Outputs:
  # Export the VPC ID with a unique name
  VPCId:
    Description: The ID of the VPC
    Value: !Ref MyVPC
    Export:
      Name: !Sub "${AWS::StackName}-VPCId" # Ensures a unique export name per stack

  # Export the Public Subnet ID with a unique name
  PublicSubnetId:
    Description: The ID of the public subnet
    Value: !Ref MyPublicSubnet
    Export:
      Name: !Sub "${AWS::StackName}-PublicSubnetId"
```

Key parts in network-stack.yaml:

- **Outputs section:** This is where you define values that other stacks can reference.
- **Export: Name:**: This property is crucial. It gives a unique name to the output value that can be imported by other stacks.
  - We use !Sub "${AWS::StackName}-VPCId" to create a unique export name that includes the name of the CloudFormation stack itself. This helps prevent naming conflicts if you deploy multiple instances of this network stack.

2. Application Stack (application-stack.yaml) - The Importer

This stack deploys an EC2 instance and **imports** the PublicSubnetId and VpcId from the network-stack.

YAML

```
# application-stack.yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Deploys an EC2 instance into an existing network using cross-stack references.
```

```
Parameters:
  # Parameter to get the name of the network stack from which to import values
  NetworkStackName:
    Type: String
    Description: The name of the CloudFormation stack that exports the network details.

Resources:
  WebServerSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupName: WebAppSG
      GroupDescription: Enable HTTP access to web server
      # Import the VPC ID from the network stack
      VpcId: !ImportValue !Sub "${NetworkStackName}-VPCId"
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 80
          ToPort: 80
          CidrIp: 0.0.0.0/0
      Tags:
        - Key: Name
          Value: WebServerSecurityGroup

  WebServerInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-0abcdef1234567890 # Replace with a valid AMI ID for your region (e.g., Amazon Linux 2 AMI)
      InstanceType: t2.micro
      # Import the Public Subnet ID from the network stack
      SubnetId: !ImportValue !Sub "${NetworkStackName}-PublicSubnetId"
      SecurityGroupIds:
        - !Ref WebServerSecurityGroup
      Tags:
        - Key: Name
          Value: MyAppWebServer

Outputs:
  WebServerPublicIp:
    Description: The public IP address of the web server
    Value: !GetAtt WebServerInstance.PublicIp
```

Key parts in application-stack.yaml:

- **Parameters section:** The NetworkStackName parameter allows you to specify which network stack to import from. This makes the application stack reusable across different environments or network deployments.
- **!ImportValue intrinsic function:** This function is used to retrieve an exported output value from another stack.

- !ImportValue !Sub "${NetworkStackName}-VPCId": This line dynamically constructs the export name (e.g., MyNetworkStack-VPCId) using the NetworkStackName parameter and then imports the corresponding VPC ID.
- Similarly for SubnetId.

How to deploy:

1. **Deploy network-stack.yaml first.** Give it a name, for example, MyNetworkStack. CloudFormation will create the VPC, subnet, etc., and then export the VPCId and PublicSubnetId with names like MyNetworkStack-VPCId and MyNetworkStack-PublicSubnetId.
2. **Deploy application-stack.yaml next.** When prompted for the NetworkStackName parameter, enter MyNetworkStack (or whatever name you used in step 1). The application stack will then import the VPC and subnet IDs from MyNetworkStack and use them to deploy the EC2 instance.

Advantages of Cross-Stack References:

- **Decoupling:** Stacks can be developed and managed independently. Changes to the network stack don't necessarily require changes to the application stack (unless the exported values themselves change).
- **Reusability:** The network stack can be a generic "base infrastructure" that multiple application stacks can use.
- **Modularity:** Breaks down large, complex infrastructure into smaller, more manageable units.
- **Clear Dependencies:** It's explicit which stacks depend on outputs from other stacks.

Important Considerations:

- **Region and Account:** Exported values can only be imported by stacks within the *same AWS account and AWS Region*.
- **Deletion/Updates:** You cannot delete or modify an exported output value in a stack if another stack is currently importing it. You must first update the importing stack(s) to remove the dependency before you can delete or change the exporting stack.
- **Uniqueness:** Export names must be unique within an AWS account and Region. Using !Sub "${AWS::StackName}-..." is a common best practice to ensure uniqueness.

1. What is a nested stack in AWS CloudFormation?

A. A stack that is deployed across multiple regions
B. A separate AWS account stack
C. A stack created within another stack using the **AWS::CloudFormation::Stack** resource
D. A stack that automatically rolls back upon failure
Answer: C

Why would you use a nested stack?

A. To enable multi-region deployments
B. To combine all resources into a single large template
C. To modularize infrastructure and reuse templates
D. To execute custom Lambda functions during stack creation
Answer: C

What is a cross-stack reference in CloudFormation?

A. A reference to a stack deployed in another AWS region
B. A way to export values from one stack and import into another
C. A way to replicate stacks across accounts
D. A method to store parameters in SSM
Answer: B

What keyword is used in CloudFormation to export a value from a stack?

A. OutputShare
B. Expose
C. Export
D. PublicRef
Answer: C

Which function is used in a dependent stack to bring in values from another stack?

A. !Ref
B. !GetAtt
C. !IncludeValue
D. !ImportValue
Answer: D