AWS Compute Blog

Building private cross-account APIs using Amazon API Gateway and AWS PrivateLink

by Julian Wood | on 20 MAY 2021 | in Amazon API Gateway, Amazon EC2, Amazon EC2 Container Registry, Amazon Elastic Kubernetes Service, Amazon VPC, Auto Scaling, AWS Cloud Development Kit, AWS Fargate, AWS Lambda, AWS PrivateLink | Permalink | Share

This post is written by Brian Zambrano, Enterprise Solutions Architect and Srinivasa Atta, Sr. Technical Account Manager

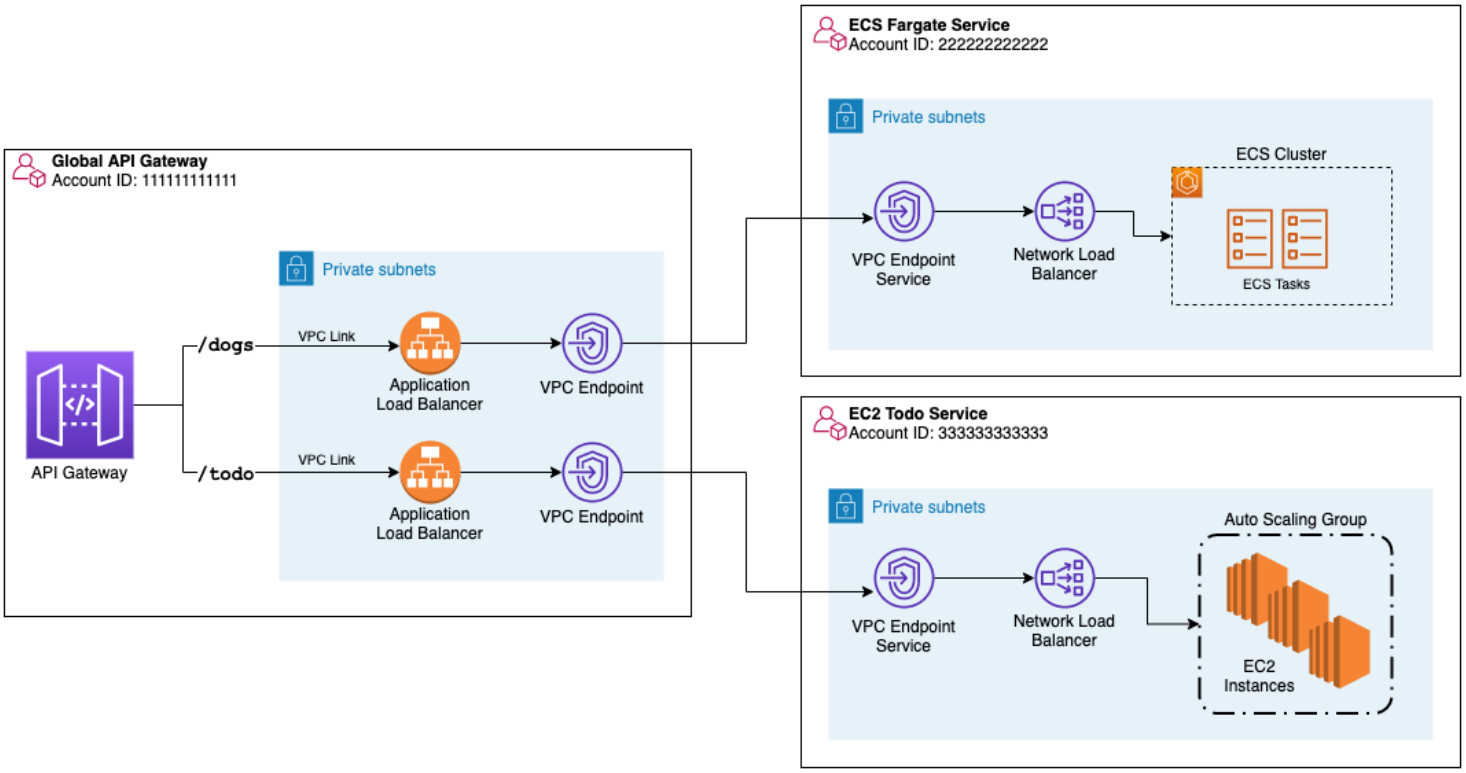
With microservice architectures, multiple teams within an organization often build different parts of an application. Different teams may own functionality for a given business segment. An effective pattern to support this is a centrally managed public API. This exposes multiple backend microservices running on various compute services such as Amazon EC2, Amazon Elastic Container Service (ECS), or Amazon Elastic Kubernetes Service.

Managing an API centrally has a number of advantages including handling authorization, authentication, and throttling in a single place. This blog post shows how to expose a single public Amazon API Gateway endpoint managed from one AWS account. It integrates securely with different backend services using VPC endpoints powered by AWS PrivateLink.

A major benefit of this approach is that network traffic stays within the Amazon network and does not traverse the public internet. This reduces attack vectors and improves the security posture. Additionally, teams can choose their own runtimes and compute platforms as long as they can expose their services as a VPC endpoint service.

Overview

The following architecture diagram provides an overview of the solution.



Solution overview

The preceding diagram depicts three services running in their own dedicated AWS accounts. The global API Gateway account (left) is referred to as the *consumer* account. Different AWS accounts (right) hosting multiple backend APIs are referred to as *provider* accounts. VPC endpoints and VPC endpoint services connect the *consumer* account to any *provider* accounts. Using PrivateLink, the traffic stays within the AWS network between API Gateway and the integration API services.

In this architecture, there are three separate AWS accounts. Each owns a specific area of functionality and responsibility.

1. **Global API Gateway: Account 111111111111**

   This is a centrally managed consumer API that provides a single HTTPS endpoint to access the two provider REST APIs. Using this architecture, you can apply authentication, authorization, rate limiting, and other API management tasks in one place.

2. **ECS Dog Names service: Account 222222222222**

   This is a read-only REST API, which returns random dog names implemented in Go, running on Amazon ECS on AWS Fargate. The Fargate service runs in private subnets and exposes its REST API via a Network Load Balancer (NLB) that is connected to a VPC Endpoint Service.

3. **EC2 ToDo service: Account 333333333333**

This is a REST API that provides a ToDo application, allowing users to create, read, and list ToDos. This service is implemented with Python using the Flask micro-framework and runs in an AWS Auto Scaling group using EC2. Like the ECS Fargate service, this application uses an NLB and VPC endpoint service.

Deploying the solution

To deploy this solution, follow the instructions in the GitHub repository and clone the repo.

You deploy this example in a single Region, us-west-2. This pattern may be applied to different Regions, but each application must exist in the *same* Region, since VPC endpoints are not supported across Regions.

Prerequisites

AWS credentials that provide the necessary permissions to create the resources with three different AWS accounts. For this example, admin credentials are used.

The resources are deployed using the AWS Cloud Development Kit (CDK) using Python. Set up CDK, install and configure Python virtual environment to proceed with deploying the solution.

In the examples, AWS environment variables are used as a mechanism to switch between different AWS accounts. For more information, see Understanding and getting your AWS credentials and configuration basics.

Deployment

The solution deploys three different systems using three different AWS accounts. The deployment, including the integration between accounts, is automated using CDK.

Create the ECS Fargate Dog Name service

The steps to deploy the ECS Fargate service include:

- Creating an Amazon Elastic Container Registry (ECR) to host the Docker image
- Build the dog-name image
- Push the image ECR
- Build the stack using CDK

1. Use AWS account 222222222222 to deploy this service and export the following environment variables:

```
Bash
export AWS_SECRET_ACCESS_KEY= ACCOUNT222222222EXAMPLEKEY
export AWS_ACCESS_KEY_ID=ACCOUNT222222222EXAMPLE
export AWS_REGION=us-west-2
export AWS_ACCOUNT=222222222222
export API_GATEWAY_ACCOUNT_ARN="arn:aws:iam::111111111111:root"
```

`AWS_SECRET_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, and `AWS_REGION` are standard environment variables, which work with the AWS CLI and CDK. `AWS_ACCOUNT` refers to the account, which owns the ECS Fargate service.

`API_GATEWAY_ACCOUNT_ARN` is a pointer to the AWS account, which connects to the VPC endpoint service.

2. Create the ECR repository:

```
Bash
aws ecr create-repository --repository-name dog-names
{
  "repository": {
    "repositoryArn": "arn:aws:ecr:AWS_REGION: 222222222222:repository/dog-names",
    "repositoryUri": "123456.dkr.ecr.us-west-2.amazonaws.com/dog-names",
    ...
  }
}
```

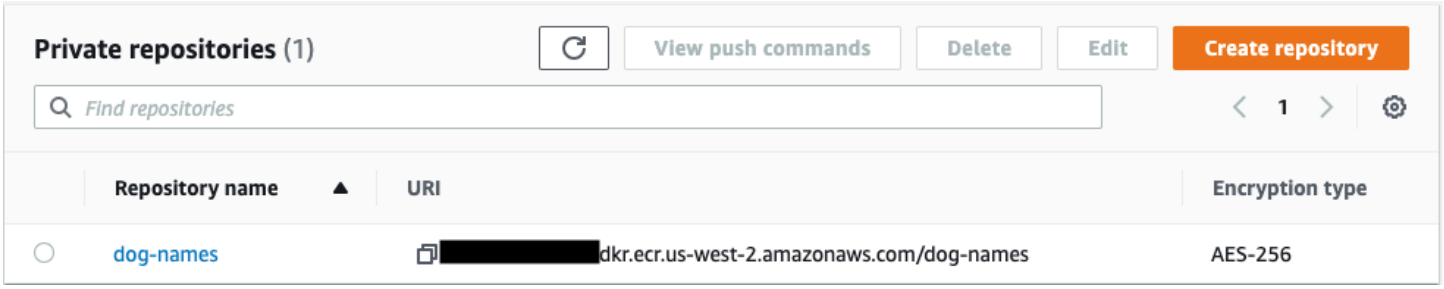
3. Build the image:

```
Bash
docker build -t dog-names:001 ./image
[+] Building 31.2s (17/17) FINISHED
...
=> => naming to docker.io/library/dog-names:001
```

4. Push the image to ECR:

```
Bash
aws ecr get-login-password | docker login \
    --username AWS \
    --password-stdin $AWS_ACCOUNT.dkr.ecr.$AWS_REGION.amazonaws.com
Login Succeeded
docker tag \
    dog-names:001 \
    $AWS_ACCOUNT.dkr.ecr.$AWS_REGION.amazonaws.com/dog-names:001
docker push \
    $AWS_ACCOUNT.dkr.ecr.$AWS_REGION.amazonaws.com/dog-names:001
```

5. Navigating to the ECR console to confirm that the *dog-names* image is successfully pushed.



dog-names repository in ECR

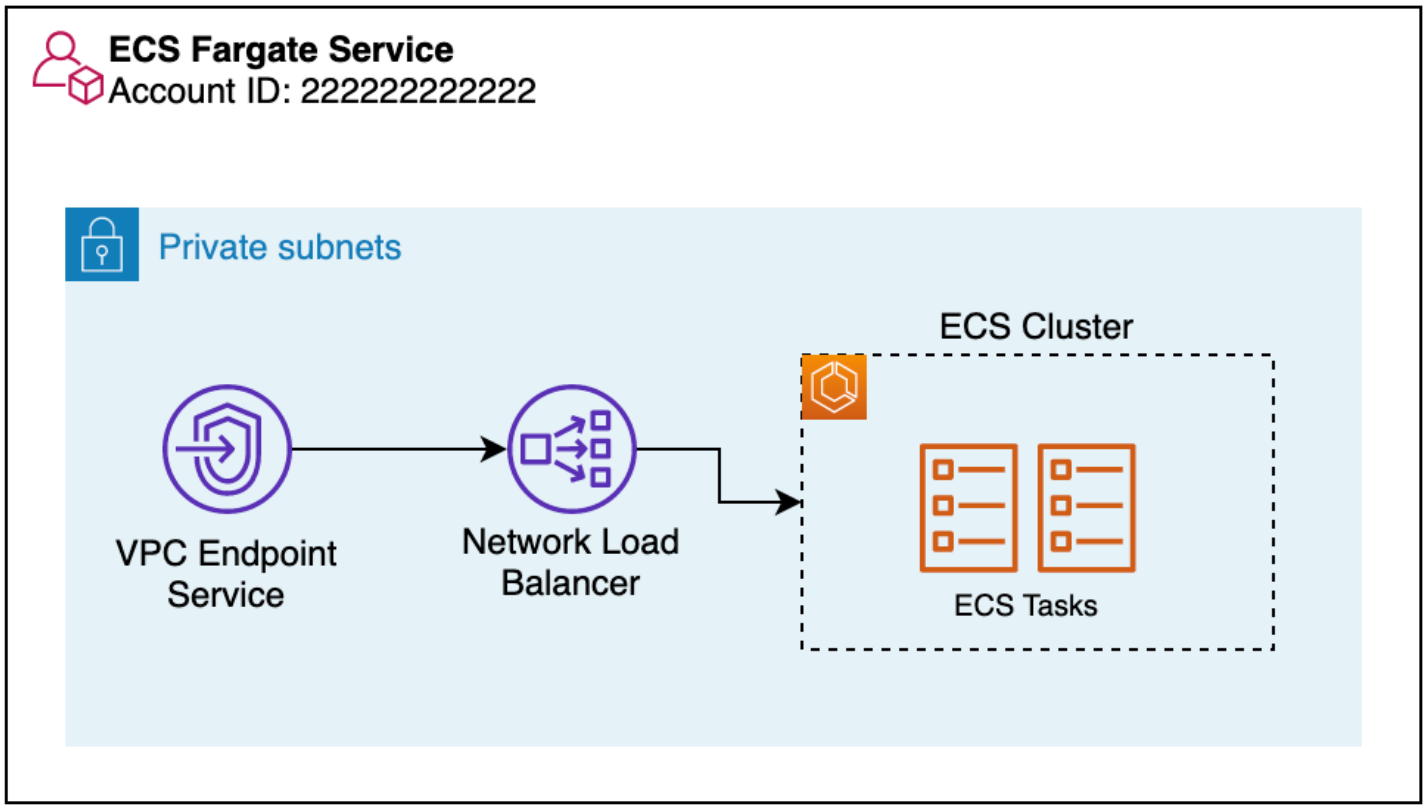
6. Now that the image is available in your account, deploy the entire stack using CDK. Enter `Y` to confirm that CDK can create or update the security groups and IAM statements.

```
Bash
cd ecs-provider
cdk deploy --all
```

CDK deploys the application using CloudFormation stacks and shows the outputs similar to the following:

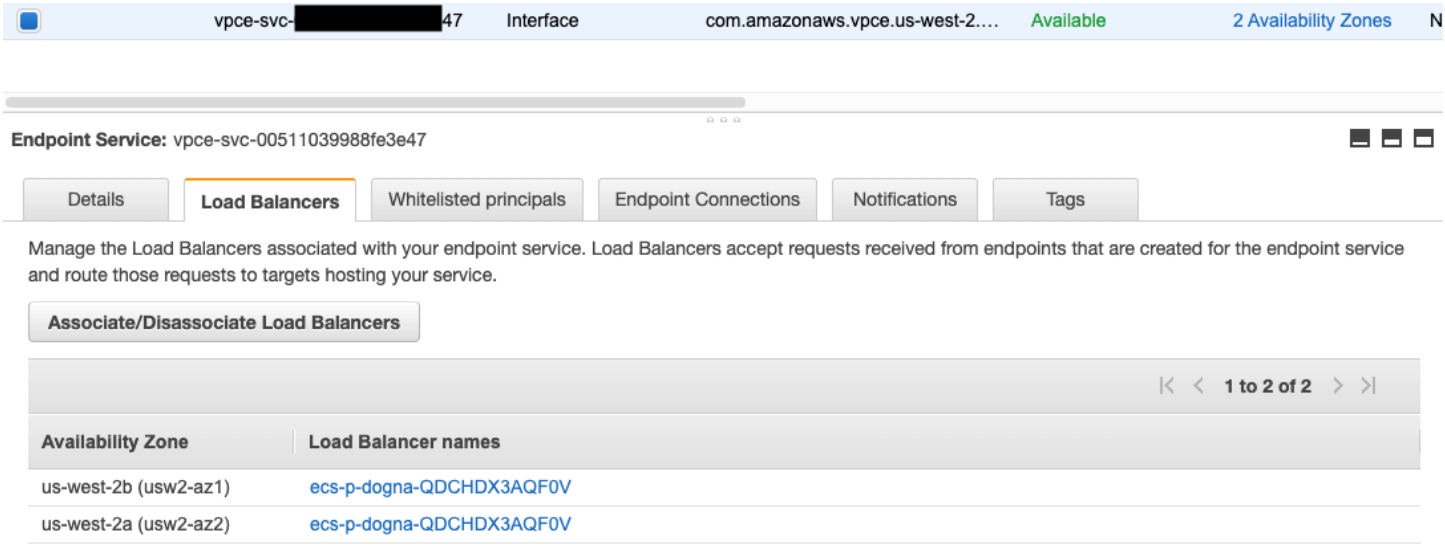
```
Bash
Outputs:
ecs-provider.dognamesLoadBalancerDNS803DE242 = ecs-p-dogna-QDCHDX3AQF0V-xxxxxxxxxxxx
ecs-provider.ecsvpcendpointservicename = com.amazonaws.vpce.us-west-2.vpce-svc-00511
```

7. Note the output name `ecsvpcendpointservicename`. This is the name of the VPC endpoint service, which is required when creating the global API Gateway.



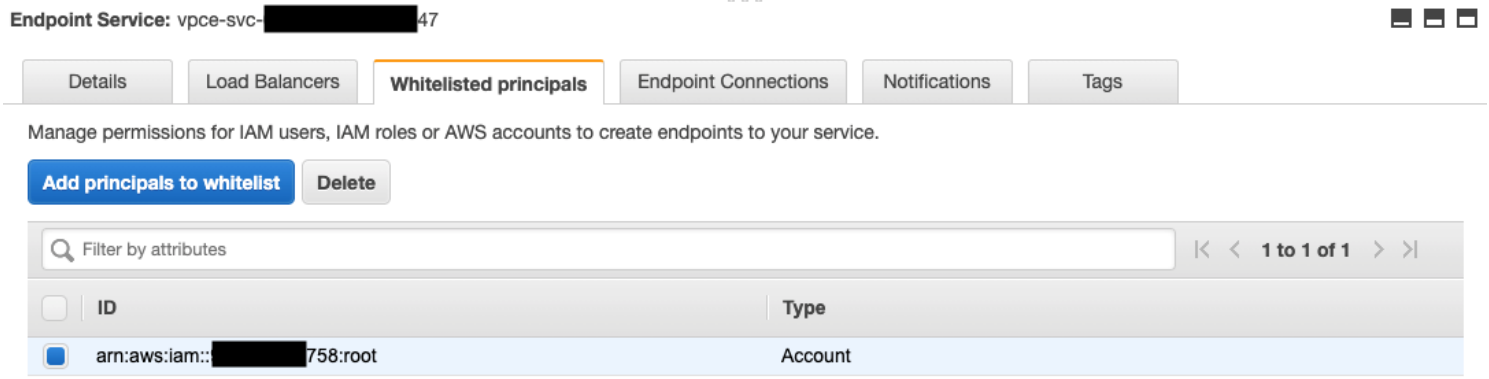
VPC endpoint service communication with ECS cluster

8. Confirm that the resources have been created. Navigate to the VPC page in the AWS Management Console and select the endpoint services link. There is a VPC endpoint service across two Availability Zones pointing to a load balancer.



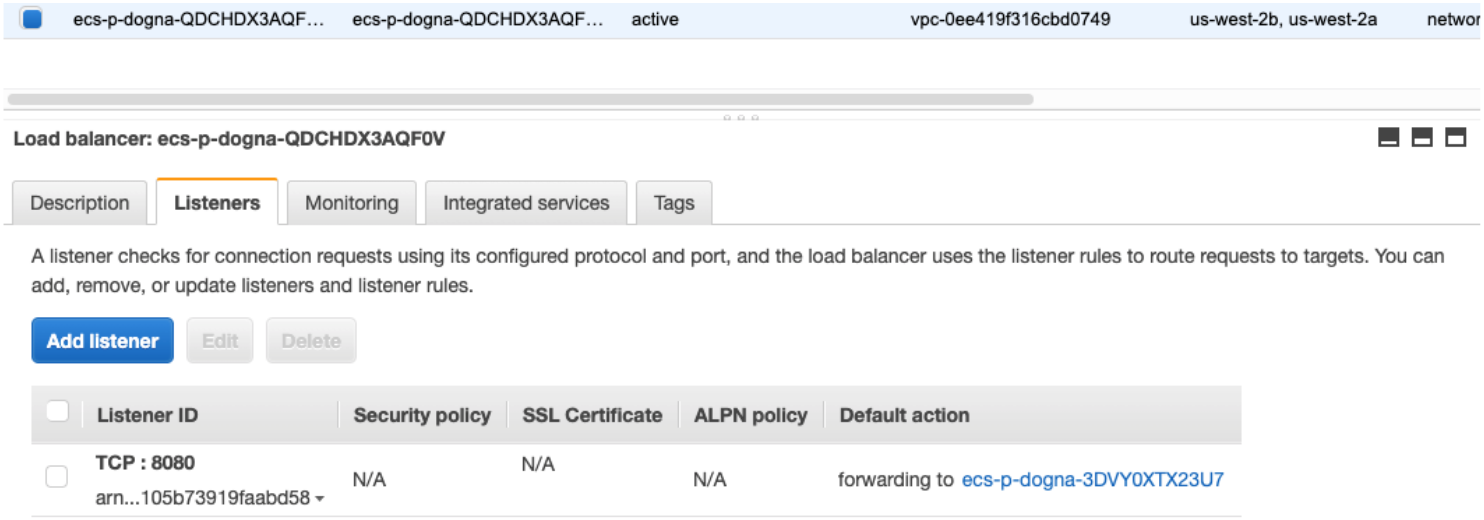
VPC endpoint service creation

9. Navigate to the allow-listed principals to see that the Global API Gateway account is added. AWS account 111111111111 is allowed to create a VPC endpoint *to* this VPC endpoint service without any additional confirmation or authorization. Without this step, the API Gateway account 111111111111 sends an authorization request that must be accepted before connecting to the Dog Name service.



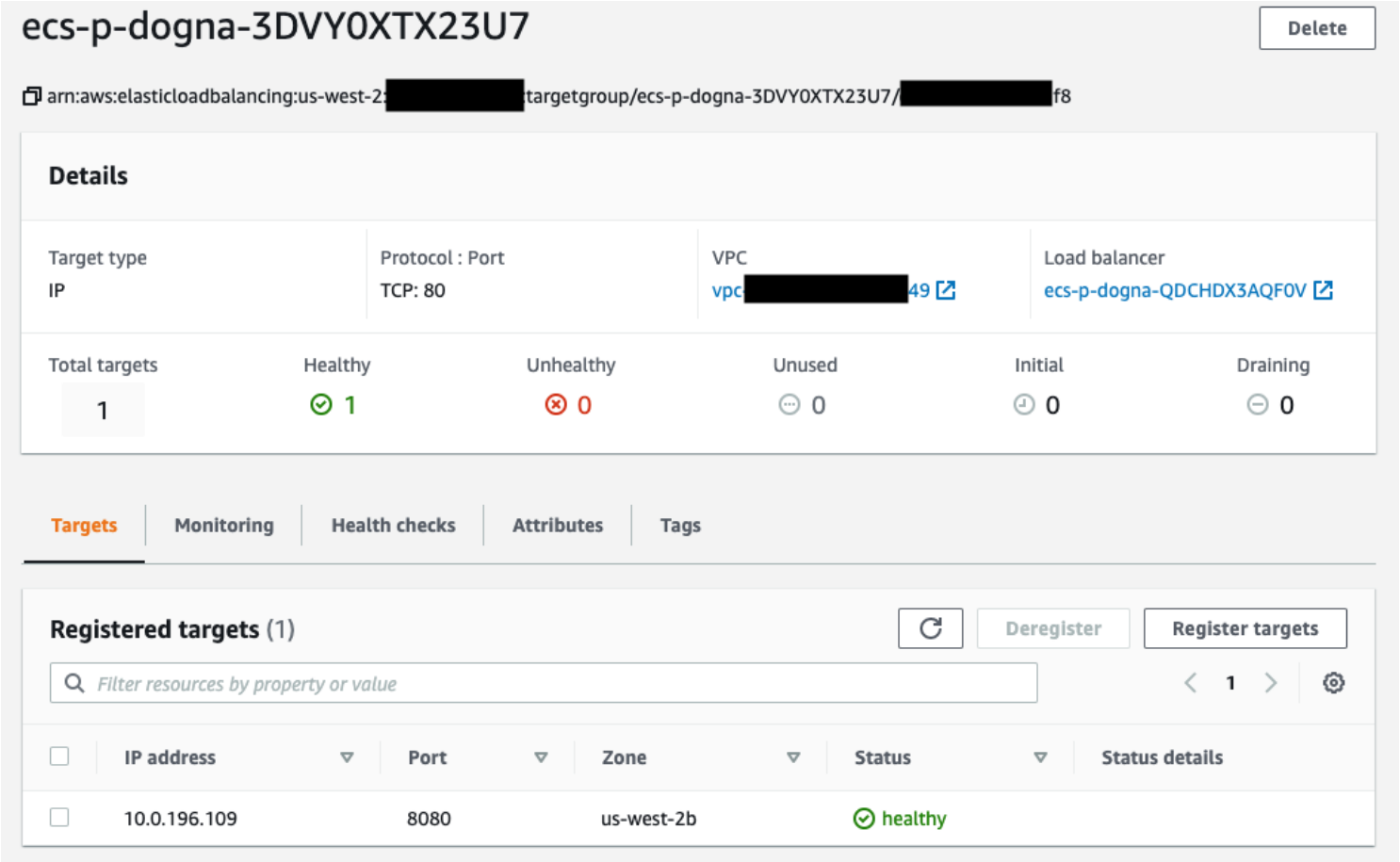
VPC endpoint service authorization request

The VPC endpoint service points to a Network Load Balancer running across two private subnets with a single HTTP listener on port 8080.



NLB running across two private subnets

This listener routes all traffic to a single private IP address, which is the Dog Name ECS task inside the ECS Fargate Cluster.



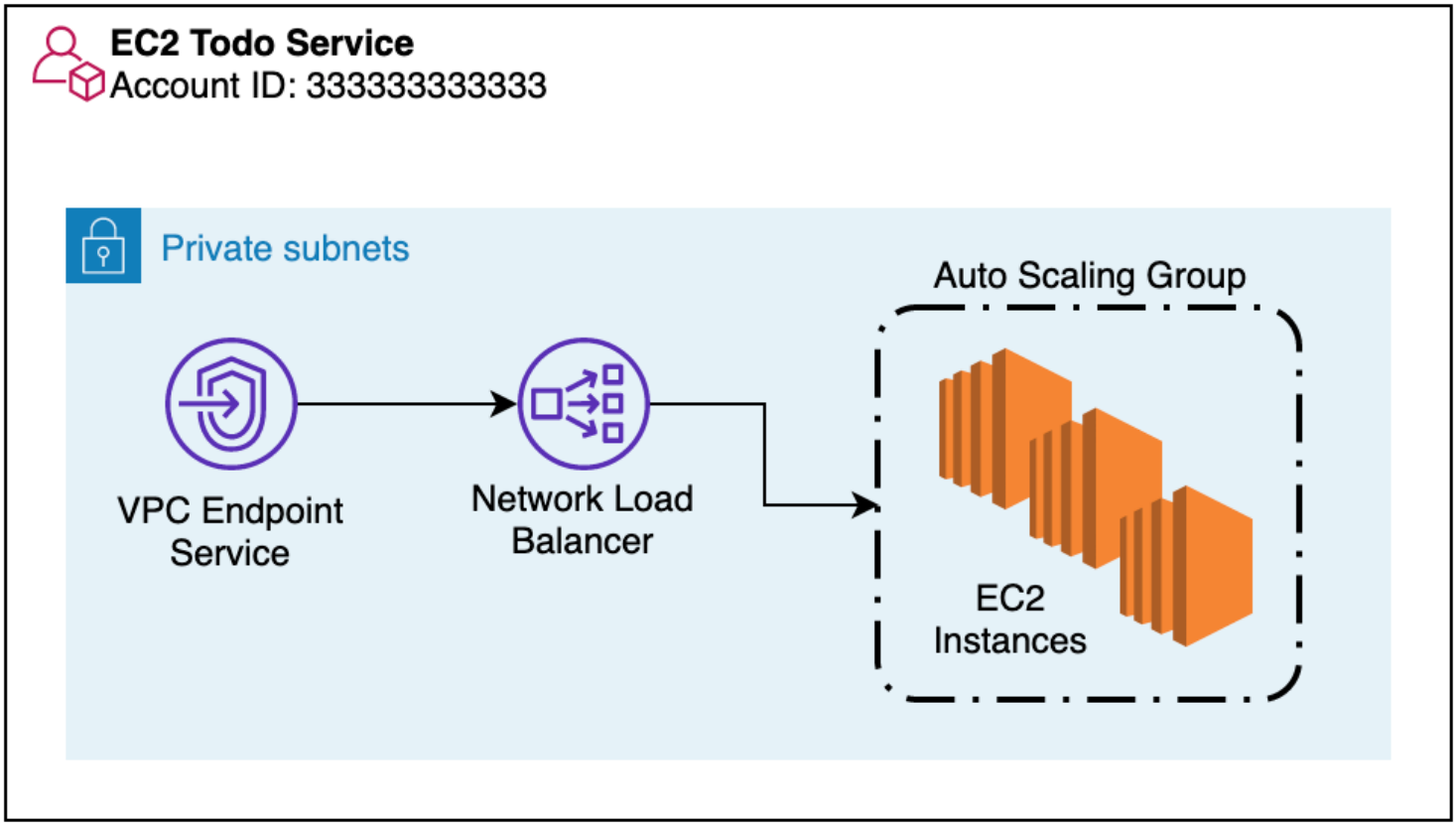
NLB routing traffic to the ECS cluster via endpoint service.

Create the EC2 Todo service

Deploy the ECS Fargate service by exporting the environment variables for the 33333333333333333 account and use CDK to deploy the stack.

```
Bash
export AWS_SECRET_ACCESS_KEY= ACCOUNT3333333333EXAMPLEKEY
export AWS_ACCESS_KEY_ID=ACCOUNT3333333333EXAMPLE
export AWS_REGION=us-west-2
export AWS_ACCOUNT=333333333333
export API_GATEWAY_ACCOUNT_ARN="arn:aws:iam::111111111:root"
cd ec2-provider
cdk deploy -all
...
Outputs:
ec2-provider.ec2vpcendpointservicename = com.amazonaws.vpce.us-east-2.vpce-svc-06acd448
ec2-provider.loadbalancer = ec2-p-todon-1NF5W2NDWPM88-xxxxxxxxxxxxxxxx.elb.us-east-2.ar
```

When the deployment completes, the stack is similar to the ECS Dog Name service, the service running on EC2 instances with an Auto Scaling group.



VPC endpoint service communication with EC2 cluster

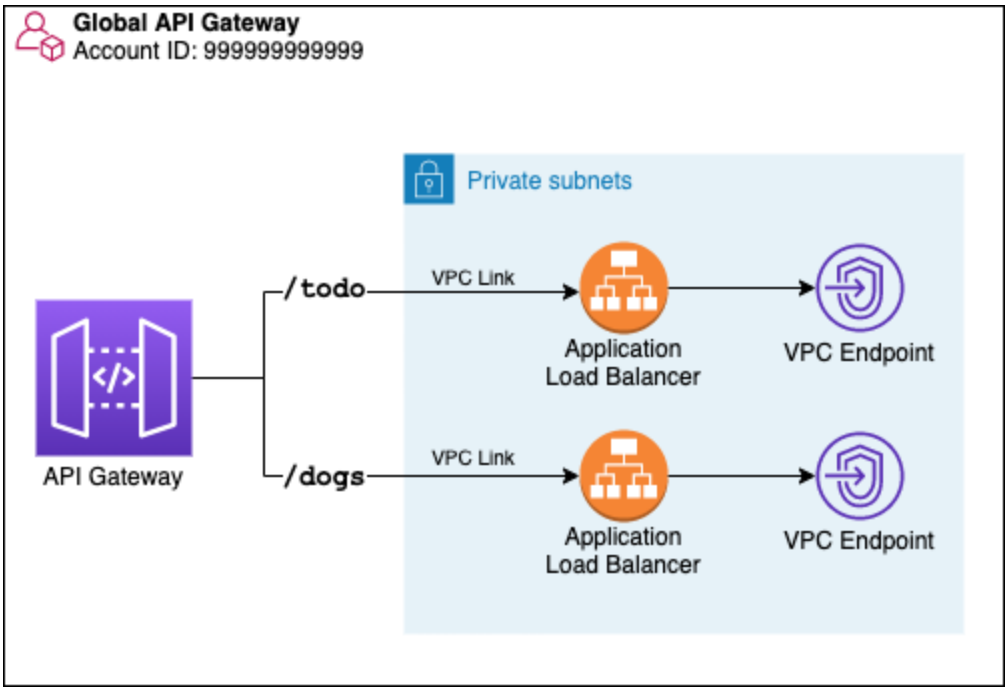
Make note of the output named `ec2vpcendpointservicename`.

Deploy the global API Gateway

With the two backing services deployed, deploy the API Gateway endpoint, which provides public access to the two services. Specify the two VPC endpoint service names from the outputs of the previous deploys.

```bash
export AWS_SECRET_ACCESS_KEY= ACCOUNT1111111111EXAMPLEKEY
export AWS_ACCESS_KEY_ID=ACCOUNT1111111111EXAMPLE
export AWS_REGION=us-west-2
export AWS_ACCOUNT=111111111111
export TODO_SERVICE_NAME=com.amazonaws.vpce.us-east-2.vpce-svc-06acd448b23628b83
export DOGNAME_SERVICE_NAME=com.amazonaws.vpce.us-west-2.vpce-svc-00511039988fe3e47
cd global-apigw
cdk deploy -all
```

Once deployed, the stack includes a single public HTTP API Gateway. It routes to the two separate services over their VPC endpoint services, depending on the path in the URL.



Global API Gateway communicating with VPC endpoints

Find the HTTP endpoint in the CDK output, which is in the following format:

```
global-apigw.HttpApiEndpoint = https://756ua49vzl.execute-api.us-west-2.amazonaws.com/
```

Testing

Test the Dog Name API by using curl, replace the URL with the HTTP endpoint from the CDK output.

Bash

```
# Get a random dog name
curl https://756ua49vzl.execute-api.us-west-2.amazonaws.com/dogs
```

Test the ToDo API by using curl. Replace the URL with the HTTP endpoint from the CDK output.

Bash

```
# Create a new Todo
curl -d data="Write new AWS blog post" https://756ua49vzl.execute-api.us-west-2.amazon
{"id": "d2be7fe9", "data": "Write new AWS blog post"}
# List Todos
curl https://756ua49vzl.execute-api.us-west-2.amazonaws.com/todo
[{"id": "d2be7fe9", "data": "Write new AWS blog post"}]
```

Conclusion

This post demonstrates how to connect securely from API Gateway to private resources in different AWS accounts by using VPC endpoints powered by PrivateLink. PrivateLink traffic does not leave the AWS network, which reduces the attack surface and decreases network latency.

This pattern can be extended to other provider resources including Amazon Elastic Kubernetes Service (EKS), on-premises workloads, or any other compute environment where a Network Load Balancer can be used.

You can centrally apply features including throttling, authentication, and authorization via API Gateway, allowing application teams to focus on their business domains.

To try out the solution, follow the instructions in the GitHub repository.

For more serverless learning resources, visit https://serverlessland.com.

TAGS: contributed, serverless