COME CREARE UNA PIPELINE DI CONTINUOUS DEPLOYMENT SU AWS PER IL DEPLOY BLUE/GREEN SU ECS.

Amazon ECS | AWS CodeBuild | AWS CodeDeploy | AWS CodePipeline | blue/green deployment | Continuous Delivery

beSharp | 30 Ottobre 2020

La **Continuous Delivery** è oggi una delle più note metodologie di rilascio del software. Grazie ad essa, qualsiasi commit che abbia superato la fase di test, potrà essere rilasciato e distribuito in produzione in modo automatico.

Automatizzare le fasi di rilascio permette, tra gli altri vantaggi, di ottenere un flusso di lavoro impeccabile negli ambienti di sviluppo, test e produzione rendendo le distribuzioni semplici e sicure.

Nel nostro ultimo articolo abbiamo parlato dei microservizi, dei loro vantaggi e di come configurare una distribuzione di tipo **blue/green su AWS** per un servizio ECS.

Riassumendo, con la tecnica del blue/green deployment la vecchia infrastruttura (blue) coesiste temporaneamente con la nuova infrastruttura (green). Dopo aver effettuato i test di integrazione/validazione, l'infrastruttura aggiornata verrà promossa a produzione, il traffico verrà reindirizzato con un processo totalmente seamless e la vecchia infrastruttura sarà eliminata definitivamente.

Come promesso, con questo articolo faremo un passo avanti: vi mostreremo come automatizzare il processo definendo una **pipeline di Continuous Deployment** che, da un semplice git push, sarà in grado di gestire in autonomia l'intero flusso di rilascio di un nuovo pacchetto software in modalità blue/green su ECS.

Infine, come bonus, mostreremo come **automatizzare i test** sugli ambienti "green" e come semplificare la **creazione di un boilerplate** complesso di infrastruttura sfruttando il servizio AWS CloudFormation. Presto capiremo come potrebbe esserci utile.

Siete pronti? Si comincia!

Prerequisiti

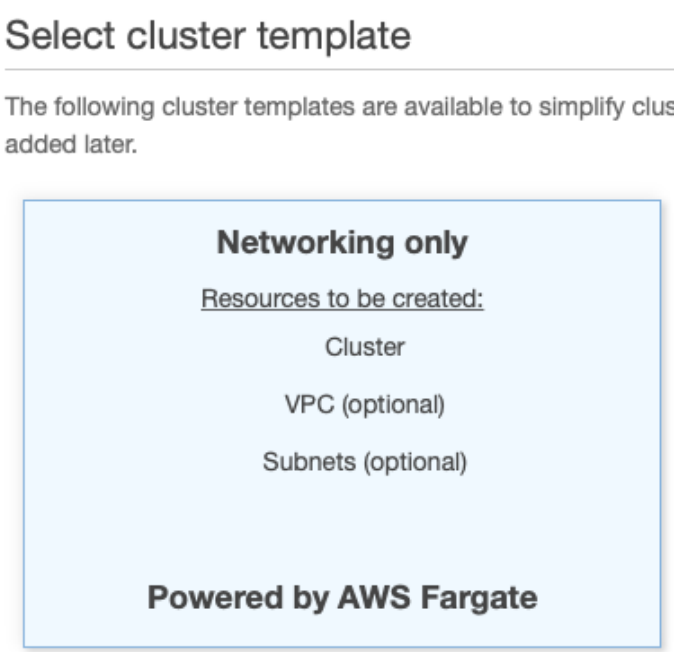
Prima di entrare nel vivo della creazione della pipeline, occorre assicurarsi che tutti questi requisiti siano soddisfatti:

- Possedere un Repository GitHub funzionante su cui poter salvare il codice, il trigger per la nostra pipeline.
- Possedere un ruolo con permessi che permettano al servizio CodeDeploy di accedere alle istanze target.
- Avere un'immagine Docker pronta con una semplice app express per ECS.
- Preparare un cluster ECS, un servizio ECS e una Task Definition sull'account AWS.

Per quest'ultimo prerequisito riassumiamo di seguito gli step che abbiamo descritto più dettagliatamente nell'articolo precedente. Se volete seguire la guida completa, leggete qui prima di proseguire.

Creare un nuovo Cluster ECS

Spostiamoci sull'account AWS e cerchiamo ECS utilizzando la barra di ricerca disponibile. Selezioniamo "Clusters" nel pannello di sinistra e, nella finestra successiva, clicchiamo su "Create Cluster". Siccome utilizzeremo Fargate, manteniamo "Networking only" come opzione e clicchiamo su "next".

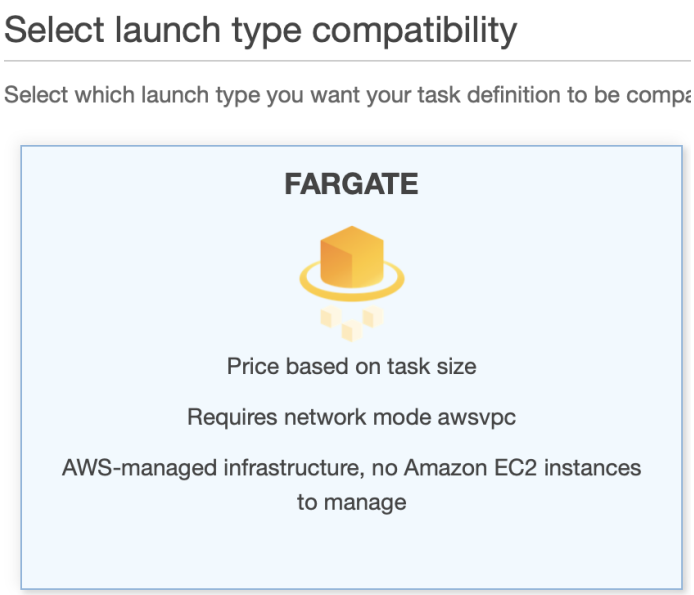


Inseriamo un nome per il cluster che stiamo creando lasciando invariate le altre opzioni e, se vogliamo, aggiungiamo alcuni tag significativi. Clicchiamo su "Create" per generare il nuovo cluster.

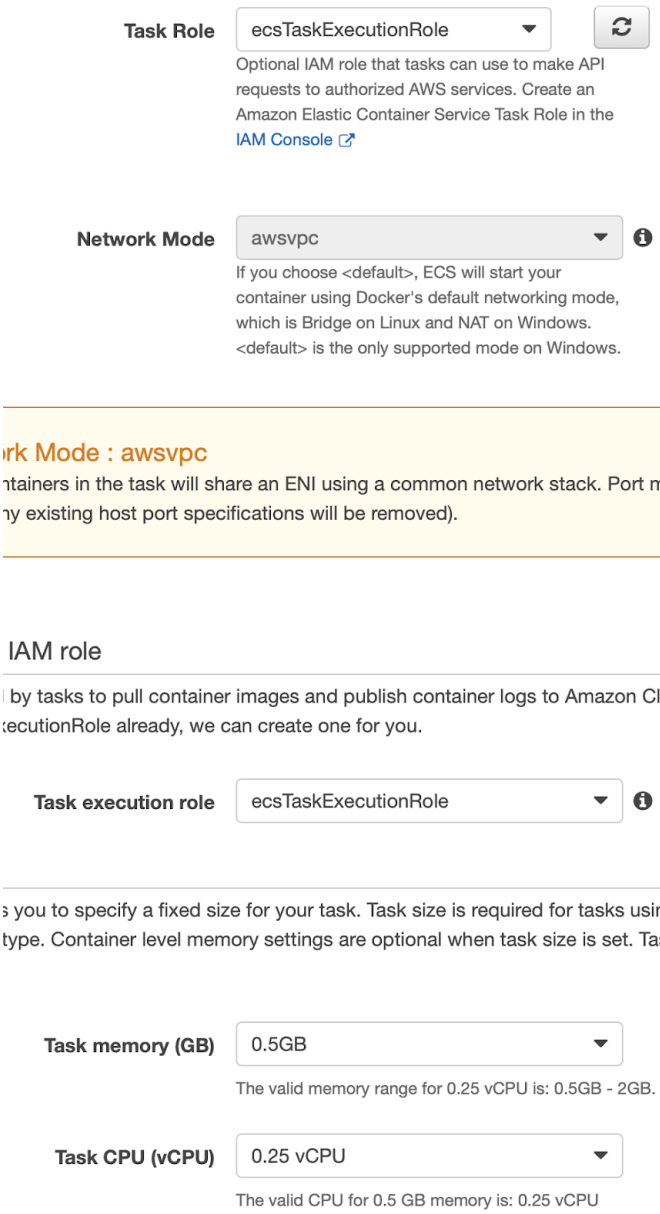
Creare una nuova Task Definition

Occupiamoci ora della Task Definition che ospiterà i nostri container Docker.

Andiamo su “Task Definitions” sotto al menu “Amazon ECS”, clicchiamo su “Create new Task Definition” e selezioniamo “Fargate” come mostrato nell’immagine. Clicchiamo poi su “Next Step”.



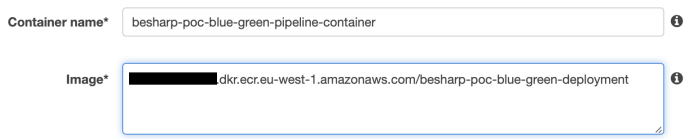
Per il momento possiamo assegnare i ruoli di default sia al Task Role, che al Task Execution Role. Per le operazioni che andremo ad eseguire saranno sufficienti. Selezioniamo i valori minimi per Memoria e CPU (per questo esempio, 0.5GB e 0.25vCPU).



Creiamo ora un Container e associamolo all'immagine Docker che abbiamo salvato su ECR (lo abbiamo spiegato qui) configurando vCPU e memoria con gli stessi valori che abbiamo indicato nella Task Definition.

Selezioniamo "Add Container".

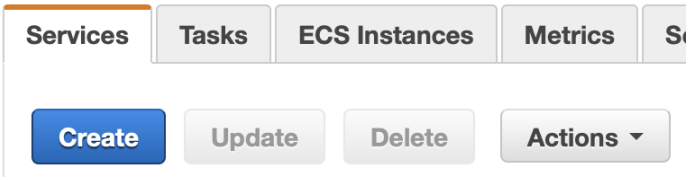
Nella sidebar impostiamo un nome per il container e per l'**Image Uri**, apriamo una nuova Tab a spostiamoci sulla dashboard di ECR. Selezioniamo l'immagine creata in precedenza e copiamone l'URL per inserirlo nel campo "Image URI".



A questo punto, aggiungiamo **3000** per **tpc protocol** in "Port mapping" e lasciamo le altre opzioni invariate prima di cliccare su "Add".

Creare un nuovo Service

Muoviamoci nel Cluster creato nel servizio ECS, clicchiamo sul suo nome e, sul fondo della dashboard, clicchiamo su “Create” sotto la tab “Services”.



Configuriamo le opzioni così come mostrato:

1. Launch Type: **FARGATE**

2. Task Definition: **<YOUR_TASK_DEFINITION>**

3. Cluster: **<YOUR_CLUSTER>**

4. Service Name: **<A_NAME_FOR_THE_SERVICE>**

5. Number of Tasks: **1**

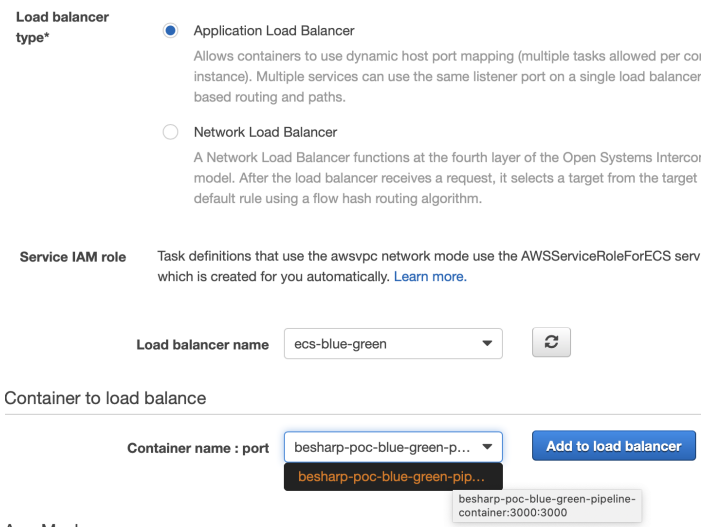
6. Deployment Type: **Blue/Green**

7. Deployment Configuration: **CodeDeployDefault.ECSAllAtOnce**

8. Service Role for CodeDeploy: **<A_SUITABLE_ROLE_WITH_ECS_PERMISSIONS>**

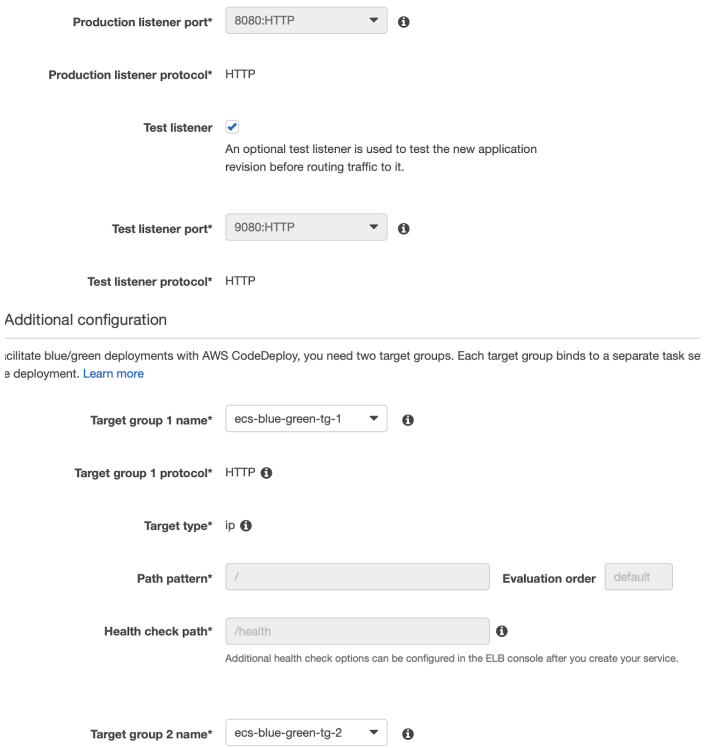
Lasciamo invariate le altre opzioni e clicchiamo su “Next Step”. Nella sezione successiva selezioniamo una VPC appropriata, una o più subnet e abilitiamo “auto-assign IP”.

Occorre ora configurare un Application LoadBalancer per il nostro Cluster. Selezioniamone uno esistente oppure creiamone uno nuovo dalla console di EC2. Scegliamo poi il nostro Container assicurandoci che mostri la porta mappata.



Dopo aver selezionato il Container, clicchiamo su “Add to load balancer”.

Indichiamo 8080 per “Production Listener Port” e 8090 per “Test Listener Port”. Selezioniamo il nostro LoadBalancer target group come mostrato in figura (se non lo avete configurato prima, potete farlo ora in una nuova tab seguendo questa guida).



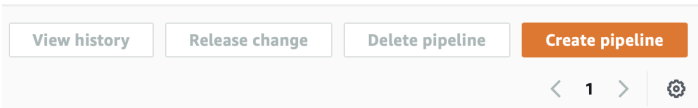
Proseguiamo lasciando spento l’autoscaling, per questo esempio non ci servirà. Dopo la revisione, il servizio è creato!

Ora che abbiamo finalmente tutti i blocchi fondamentali per la nostra Pipeline, procediamo con la creazione!

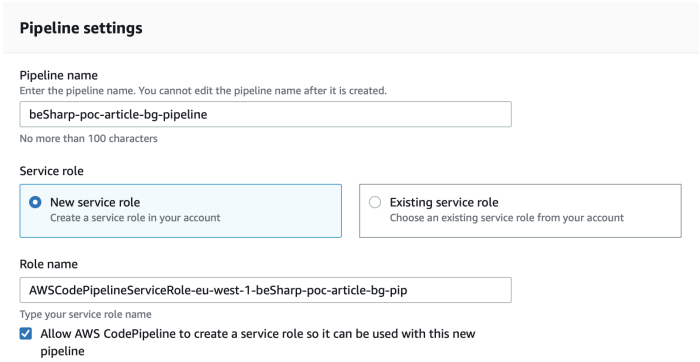
Creare la Pipeline di Deploy

Cominciamo con il “pushare” la nostra applicazione di prova sul nostro repository GitHub.

Andiamo poi sul nostro account AWS, selezionando AWS CodePipeline dalla lista dei servizi. Dalla dashboard clicchiamo su “Create pipeline”.

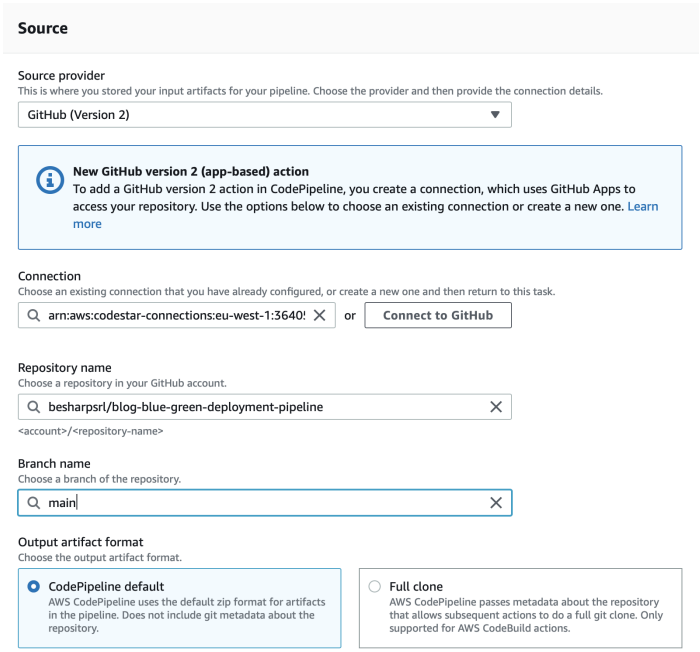


Nella schermata seguente diamo un nome alla pipeline e, se non abbiamo già un ruolo adeguato alle operazioni, lasciamo “New service role” selezionato e le altre opzioni come default; clicchiamo su “next”.



Nel **source** stage selezioniamo “GitHub version 2” e quindi connettiamoci al nostro repository GitHub. Seguiamo le istruzioni che ci verranno presentate dopo aver selezionato “Connect to GitHub”. Ricordiamoci di **autorizzare solo il repository della nostra soluzione** e di eseguire le operazioni come **owner di quel repository,** altrimenti non saremo in grado di completare il processo.

Dopo esserci connessi a GitHub, potremo completare i passaggi come mostrato di seguito, selezionando repository e branch:



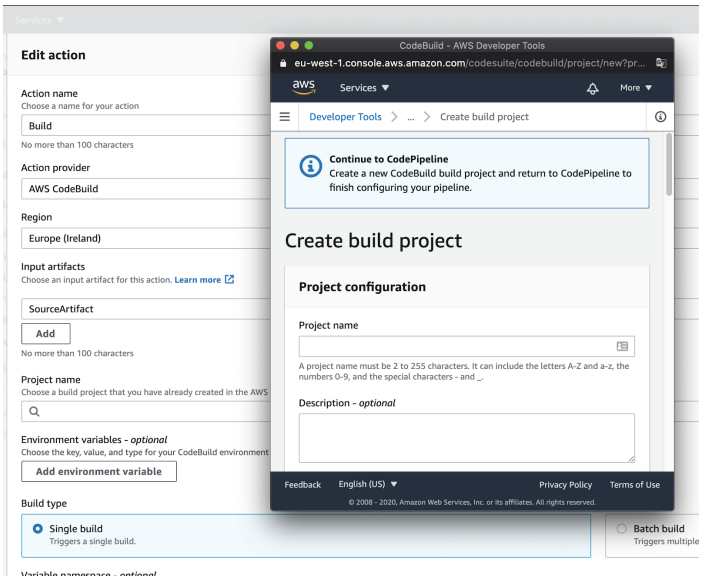
Clicchiamo “next” e potremo proseguire con lo stage di build dove creeremo il nostro progetto CodeDeploy da aggiungere alla pipeline.

Creare un nuovo progetto di CodeBuild

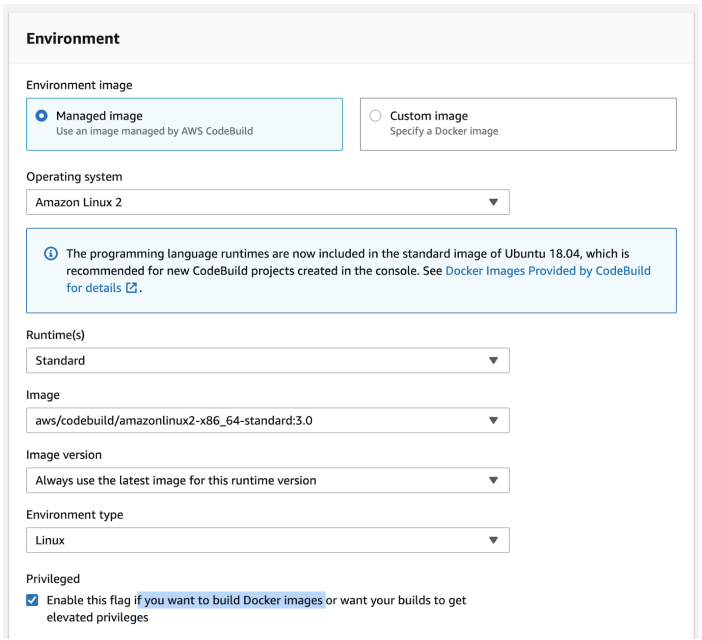
Per mantenere il codice sempre aggiornato nella nostra pipeline, abbiamo bisogno di questo step per generare un’immagine sempre aggiornata di Docker.

Cominciamo col dare un nome al nostro **stage di Build**, selezioniamo **CodeBuild** come “Action provider”, la region e **SourceArtifact** come “Input Artifact”.

Dobbiamo ora creare un nuovo progetto di build.
Cliccando su “Add project”, ci troveremo davanti ad una schermata simile a questa:



Diamo un nome al progetto, quindi lasciamo **Managed Image** con tutte le proprietà del container come suggerito. Proseguiamo **cliccando** (questo è estremamente importante) su “Privileged option” per permettere alla pipeline di generare le immagini Docker per noi. Verificate i vostri settaggi con l’immagine sottostante:



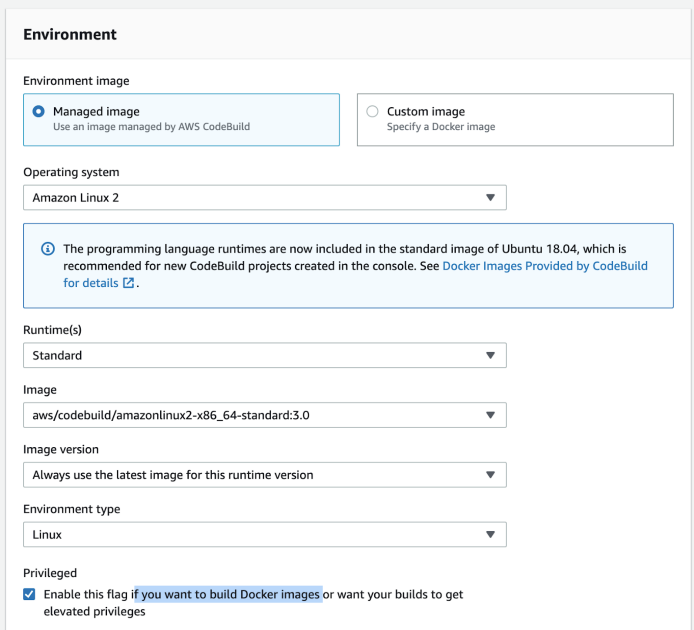
Per l’opzione di **buildspec**, selezioniamo l’editor inline e copiamo questi comandi:

```
version: 0.2
phases:
 pre_build:
   commands:
     - REPOSITORY_URI=YOUR_ECR_URI
     - echo $CODEBUILD_RESOLVED_SOURCE_VERSION
     - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION)
     - IMAGE_TAG=${COMMIT_HASH}:latest
     - $(aws ecr get-login --no-include-email --region YOUR_REGION)
 install:
   runtime-versions:
     java: corretto11
 build:
   commands:
     - printf '{"ImageURI":"%s"}' $REPOSITORY_URI:latest > imageDetail.json
```

```
      - docker build -t YOUR_ECR_URI:latest .
      - docker push YOUR_ECR_URI:latest
artifacts:
files: imageDetail.json
```

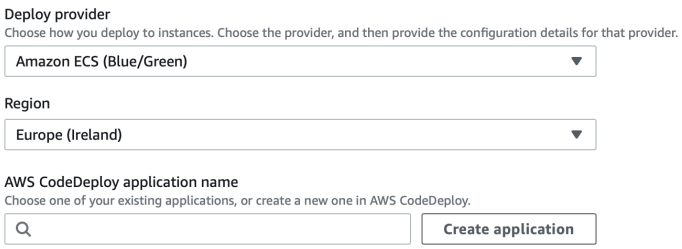
Nota: in grassetto abbiamo evidenziato le variabili che dovete adeguare al vostro progetto.

Dopo aver copiato i comandi, cliccate “ok”, quindi aggiungete il vostro progetto di Build allo stage.



Creare un nuovo progetto di CodeDeploy

Cominciamo selezionando “Amazon ECS (Blue/Green)” alla voce “Deploy Provider”, la region voluta per il progetto, quindi clicchiamo su “Create application”.



Diamo un nuovo nome al progetto e selezioniamo “Amazon ECS” come Compute Provider. Comparirà la schermata di creazione di un nuovo Development Group.

Nominiamolo, dopodiché selezioniamo, nell’ordine:

- Un service role con permessi appropriati

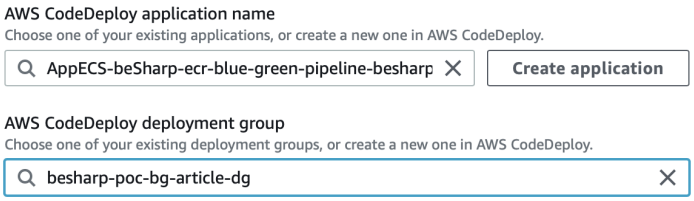
- Il cluster ECS creato in precedenza

- Il servizio ECS creato in precedenza

- L’Application Load Balancer creato prima con, rispettivamente, 8080 e TargetGroup 1 per produzione e 8090 e TargetGroup 2 per l’ambiente di test

– Una strategia per il traffico; per il nostro esempio useremo “Specify when to reroute traffic” e selezioneremo **5 minuti.**

Clicchiamo su “Create” e torniamo a CodePipeline. Selezioniamo l’applicazione **CodeDeploy** e il **CodeDeploy deployment group** appena creati.



Per “Input Artifacts” aggiungiamo **BuildArtifact** affianco a “SourceArtifact”.

Per **Amazon ECS Task Definition** e **AWS CodeDeploy AppSpec file** selezioniamo “Source Artifact”, dopodiché aggiungiamo BuildArtifact e IMAGE come ultime opzioni. Clicchiamo su “Next”, facciamo la review e clicchiamo infine su “Create pipeline”.

Ci siamo quasi: per completare la nostra pipeline abbiamo bisogno di aggiungere una **task definition** e un **appspec.yml** alla nostra applicazione.

Creiamo dunque un nuovo file **appspec.yml** nella root del progetto e aggiungiamo al suo interno questo codice:

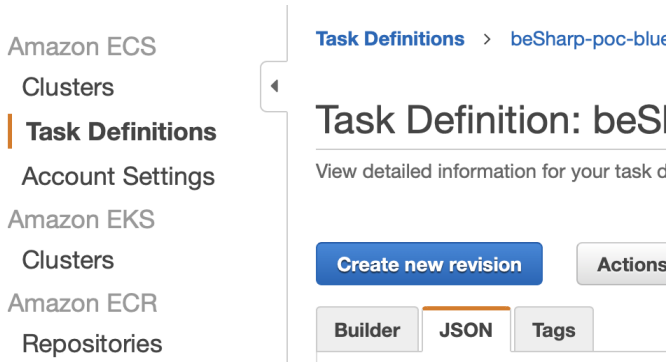
```
version: 0.0
Resources:
 - TargetService:
     Type: AWS::ECS::Service
     Properties:
       TaskDefinition: "<TASK_DEFINITION>"
       LoadBalancerInfo:
         ContainerName: "YOUR_ECS_CLUSTER_NAME"
         ContainerPort: 3000
```

Passiamo ora al file task definition.

Per questo useremo un trucco per semplificarne la creazione. Come ricorderete, abbiamo già creato una task definition in fase di preparazione dei prerequisiti all’inizio di questo articolo. Andiamo a recuperare quest’ultima da AWS e clicchiamo su “Edit”. Arriveremo all’editor JSON. Copiamo il testo e incolliamolo in un nuovo file **taskdef.json** all’interno della root del nostro progetto. Fatto ciò, andiamo a modificare così le seguenti righe:3.”image”: “<IMAGE>” **(rimuovere URL e mettere <Image>)**

```
"image": "<IMAGE>"
"taskDefinitionArn": "<TASK_DEFINITION>"
```

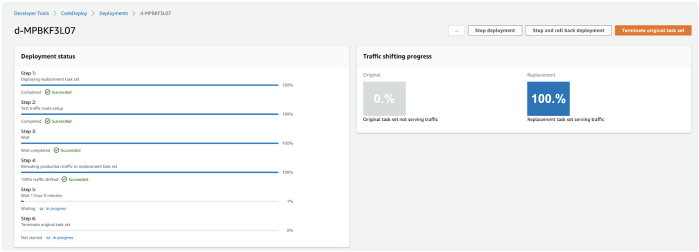
Carichiamo la nuova versione del codice sul nostro repo.



Testiamo l'applicazione prima di promuoverla a Produzione

Per verificare che tutto funzioni correttamente, apportiamo una piccola modifica al testo contenuto nella root principale dell'applicazione, facciamo un commit e aspettiamo che la Pipeline esaurisca tutti i task. A questo punto, verifichiamo che solo l'URL sulla porta 8090 riporti la modifica di test. Sulla porta 8080 dovremmo invece continuare a vedere la vecchia versione dell'infrastruttura. Dopo 5-6 minuti (il tempo impostato per il routing del traffico), anche l'ambiente di produzione dovrebbe aggiornarsi mostrando la versione corretta.

La nostra Pipeline funziona!



Bonus 1: automatizzare i test sull'infrastruttura *green* con AWS Lambda

Nella fase di rilascio è possibile associare una o più Lambda function col compito di verificare il buon funzionamento della nostra applicazione prima di promuovere la nuova versione in un ambiente di produzione. Questo procedimento lo si svolge durante la configurazione del Deployment lifecycle hooks. Occorrerà solo aggiungere un Lambda hook a AfterAllowTraffic.

Seguite queste guide per un esempio di configurazione:

- https://docs.aws.amazon.com/codedeploy/latest/userguide/tutorial-ecs-deployment-with-hooks.html

- https://docs.aws.amazon.com/codedeploy/latest/userguide/tutorial-ecs-with-hooks-create-hooks.html

Bonus 2: creiamo un template AWS CloudFormation per

automatizzare la predisposizione dei prerequisiti.

Uno dei prerequisiti necessari consisteva nella creazione di un cluster ECS e dei suoi componenti. Di tutta la configurazione, questo è sicuramente uno dei passaggi più complessi... ma che possiamo semplificare e addirittura rendere ripetibile! Come?

Creando un CloudFormation template!

Ecco un semplice snippet di esempio che vi aiuterà ad impostarlo:

```
LoadBalancer:
   Type: AWS::ElasticLoadBalancingV2::LoadBalancer
   Properties:
     Name: !Ref ProjectName
     LoadBalancerAttributes:
       - Key: 'idle_timeout.timeout_seconds'
         Value: '60'
       - Key: 'routing.http2.enabled'
         Value: 'true'
       - Key: 'access_logs.s3.enabled'
         Value: 'true'
       - Key: 'access_logs.s3.prefix'
         Value: loadbalancers
       - Key: 'access_logs.s3.bucket'
         Value: !Ref S3LogsBucketName
       - Key: 'deletion_protection.enabled'
         Value: 'true'
       - Key: 'routing.http.drop_invalid_header_fields.enabled'
         Value: 'true'
     Scheme: internet-facing
     SecurityGroups:
       - !Ref LoadBalancerSecurityGroup
     Subnets:
       - !Ref SubnetPublicAId
       - !Ref SubnetPublicBId
       - !Ref SubnetPublicCId
     Type: application
 HttpListener:
   Type: AWS::ElasticLoadBalancingV2::Listener
   Properties:
     DefaultActions:
       - RedirectConfig:
           Port: '443'
           Protocol: HTTPS
           StatusCode: 'HTTP_301'
         Type: redirect
     LoadBalancerArn: !Ref LoadBalancer
     Port: 80
     Protocol: HTTP
 HttpsListener:
   Type: AWS::ElasticLoadBalancingV2::Listener
   Properties:
     Certificates:
```

```
      - CertificateArn: !Ref LoadBalancerCertificateArn
    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref TargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Port: 443
    Protocol: HTTPS
TargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    Name: !Ref ProjectName
    HealthCheckIntervalSeconds: 30
    HealthCheckPath: !Ref HealthCheckPath
    HealthCheckProtocol: HTTP
    HealthCheckPort: !Ref NginxContainerPort
    HealthCheckTimeoutSeconds: 10
    HealthyThresholdCount: 2
    UnhealthyThresholdCount: 2
    Matcher:
      HttpCode: '200-299'
    Port: 8080
    Protocol: HTTP
    TargetType: ip
    TargetGroupAttributes:
      - Key: deregistration_delay.timeout_seconds
        Value: '30'
    VpcId: !Ref VpcId
Cluster:
  Type: AWS::ECS::Cluster
  Properties:
    ClusterName: !Ref ProjectName
Service:
  Type: AWS::ECS::Service
  Properties:
    Cluster: !Ref Cluster
    DeploymentConfiguration:
      MaximumPercent: 200
      MinimumHealthyPercent: 100
    DesiredCount: 3
    HealthCheckGracePeriodSeconds: 60
    LaunchType: FARGATE
    LoadBalancers:
      - ContainerName: ContainerOne
        ContainerPort: !Ref ContainerPort
        TargetGroupArn: !Ref TargetGroup
    NetworkConfiguration:
      AwsvpcConfiguration:
        AssignPublicIp: DISABLED
        SecurityGroups:
          - !Ref ContainerSecurityGroupId
        Subnets:
          - !Ref SubnetPrivateNatAId
          - !Ref SubnetPrivateNatBId
          - !Ref SubnetPrivateNatCId
    ServiceName: !Ref ProjectName
    TaskDefinition: !Ref TaskDefinition
  DependsOn: HttpsListener
TaskDefinition:
  Type: AWS::ECS::TaskDefinition
  Properties:
```

```
      Family: !Ref ProjectName
      ContainerDefinitions:
        - Cpu: 2048
          Image: !Ref ContainerImageUri
          Memory: 4096
          MemoryReservation: 4096
          PortMappings:
            - ContainerPort: !Ref ContainerPort
              Protocol: tcp
          Name: ContainerOne
          LogConfiguration:
            LogDriver: awslogs
            Options:
              awslogs-group: !Ref ContainerLogGroup
              awslogs-region: !Ref AWS::Region
              awslogs-stream-prefix: ContainerOne
      Cpu: '2048'
      Memory: '4096'
      ExecutionRoleArn: !GetAtt ExecutionContainerRole.Arn
      TaskRoleArn: !GetAtt TaskContainerRole.Arn
      NetworkMode: awsvpc
      RequiresCompatibilities:
        - FARGATE
```

Questo snippet di codice è puramente informativo; dovrete adattare da voi la parte di gestione dei parametri facendo riferimento alle specifiche del vostro progetto. In caso di necessità si può fare riferimento a questi due link:

- https://github.com/awslabs/aws-cloudformation-templates/tree/master/aws/services
- https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html

Conclusioni

In questo articolo abbiamo visto come creare una Pipeline completamente automatica di Deploy in modalità Blue/Green di un servizio su ECS.

Abbiamo anche capito come le Lambda function possono essere utilizzate per automatizzare le fasi di test sull'infrastruttura "green".

Per completezza del tutorial, vi abbiamo anche mostrato come utilizzare un AWS CloudFormation template per semplificare la creazione di una infrastruttura standard complessa minimizzando i tempi e rendendola facilmente replicabile.

Il nostro intento era creare una traccia per far comprendere utilizzi e potenzialità di questa modalità automatica di rilascio del software lasciando comunque spazio per tutte le personalizzazioni necessarie a seconda dello specifico caso in cui si andrà ad applicarla.

Cosa ne pensate? Avete realizzato particolari configurazioni per le vostre Pipeline?

Siamo curiosi di scoprirlo!

Per oggi è tutto. **#Proud2beCloud** vi dà appuntamento come sempre a tra 14 giorni!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it