deployfish Documentation

Release 1.9.1

Chris Malek, Glenn Bach

Sep 07, 2022

CONTENTS:

`deployfish` has commands for managing the whole lifecycle of your application:

- Safely and easily create, update, destroy and restart ECS services
- Safely and easily create, update, run, schedule and unschedule ECS tasks
- Extensive support for ECS related services like load balancing, application autoscaling and service discovery
- Easily scale the number of containers in your service, optionally scaling its associated autoscaling group at the same time
- Manage multiple environments for your task or service (test, qa, prod, etc.) in multiple AWS accounts.
- Uses AWS Parameter Store for secrets for your containers
- View the configuration and status of running ECS services
- Run a one-off command related to your service
- Easily exec through your VPC bastion host into your running containers, or ssh into a ECS container machine in your cluster.
- Setup SSH tunnels to the private AWS resources in VPC that your service uses so that you can connect to them from your work machine.
- Extensible! Add additional functionality through custom deployfish modules.
- Works great in CodeBuild steps in a CodePipeline based CI/CD system!

Additionally, `deployfish` integrates with Terraform state files so that you can use the values of terraform outputs directly in your `deployfish` configurations.

CHAPTER

ONE

INTRODUCTION

`deployfish` has commands for managing the whole lifecycle of your application:

- Safely and easily create, update, destroy and restart ECS services
- Safely and easily create, update, run, schedule and unschedule ECS tasks
- Extensive support for ECS related services like load balancing, application autoscaling and service discovery
- Easily scale the number of containers in your service, optionally scaling its associated autoscaling group at the same time
- Manage multiple environments for your task or service (test, qa, prod, etc.) in multiple AWS accounts.
- Uses AWS Parameter Store for secrets for your containers
- View the configuration and status of running ECS services
- Run a one-off command related to your service
- Easily exec through your VPC bastion host into your running containers, or ssh into a ECS container machine in your cluster.
- Setup SSH tunnels to the private AWS resources in VPC that your service uses so that you can connect to them from your work machine.
- Extensible! Add additional functionality through custom deployfish modules.
- Works great in CodeBuild steps in a CodePipeline based CI/CD system!

Additionally, `deployfish` integrates with Terraform state files so that you can use the values of terraform outputs directly in your `deployfish` configurations.

To use `deployfish`, you

- Install `deployfish`
- Define your tasks and services in `deployfish.yml`
- Use `deploy` to start managing your tasks and services

A simple `deployfish.yml` looks like this:

```
services:
  - name: my-service
    environment: prod
    cluster: my-cluster
    count: 2
    load_balancer:
      service_role_arn: arn:aws:iam::123142123547:role/ecsServiceRole
```

(continues on next page)

(continued from previous page)

```
    load_balancer_name: my-service-elb
    container_name: my-service
    container_port: 80
  family: my-service
  network_mode: bridge
  task_role_arn: arn:aws:iam::123142123547:role/myTaskRole
  containers:
    - name: my-service
      image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/my-service:0.0.1
      cpu: 128
      memory: 256
      ports:
        - "80"
      environment:
        - ENVIRONMENT=prod
        - ANOTHER_ENV_VAR=value
        - THIRD_ENV_VAR=value
```

See the `examples/` folder in this repository for example `deployfish.yml` files.

CHAPTER

TWO

INSTALLATION

deployfish is a pure python package. As such, it can be installed in the usual python ways. For the following instructions, either install it into your global python install, or use a python virtual environment to install it without polluting your global python environment.

2.1 Install deployfish

```
pip install deployfish
```

2.2 Install AWS CLI v2

deployfish requries AWS CLI v2 for some of its functionality, notably EXEC'ing into FARGATE containers. While AWS CLI v1 was installable via *pip*, AWS CLI v2 is not, so we have to do the install manually. Here's how to set that up on a Mac:

```
# Uninstall any old versions of the cli
pip uninstall awscli

# Deactivate any pyenv environment so we can be in the system-wide Python interpreter
cd ~

# Install the new AWS CLI from brew -- it's no longer pip installable
brew update
brew install awscli

# Install the Session Manager plugin
curl "https://s3.amazonaws.com/session-manager-downloads/plugin/latest/mac/
↪sessionmanager-bundle.zip" -o "sessionmanager-bundle.zip"
unzip sessionmanager-bundle.zip
sudo ./sessionmanager-bundle/install -i /usr/local/sessionmanagerplugin -b /usr/local/
↪bin/session-manager-plugin
```

If later on you have issues with EXEC'ing or with the *aws* command in general, check to ensure you're getting your global v2 version of *aws* instead of an old v1 one from your current virtual environment:

```
aws --version
```

If the version string shows version < 2:

```
pip uninstall awscli
```

CHAPTER

THREE

TUTORIALS

3.1 A Basic Service

3.1.1 Problem

In this tutorial we will configure the bare essentials. Everything in the configuration is required. Further tutorials will look at some of the optional parameters.

The configuration below will result in a single container running in an AWS ECS cluster. The container is built from a simple nginx based hello-world image available on http://dockerhub.com, named tutum/hello-world.

3.1.2 Setup

In order to deploy this configuration, you will need an AWS ECS cluster, containing at least one EC2 machine, on which to run the container. You can either create a cluster named *hello-world-cluster* or change the *cluster* parameter in the configuration file to correspond to the name of the cluster that you created.

3.1.3 Configuration

Here's the configuration for this service:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
```

AWS ECS is made up of *services*, *tasks*, and *task definitions*. The *task definitions* define the *task* or *service*. A *task* is a container that runs and exits, while a *service* is a container that stays running, like a web server, and will be restarted by ECS if it shuts down unexpectedly.

The configuration files you will use with *deployfish* are YAML based. A typical project or application will have a single *deployfish.yml* file, containing all of the project's relevant services. This initial example only defines a single service.

If you want to define additional services, you simply have to add another name to the *services* array, along with its corresponding parameters:

```
services:
    - name: name1
      cluster: cluster1
      ...
    - name: name2
      cluster: cluster2
      ...
```

Required Service Parameters

Each *service* contains at least the five following required parameters:

name
The name of the ECS service. In this case, it is *hello-world-test*. This has to be unique.

cluster
The ECS cluster that will run the resultant container.

count
The number of containers to run, which is 1 in this case.

family
The base name of the *task definition*. Each revision of your image will have its own *task definition* consisting of the base name and the revision number. We are naming this base name *hello-world*.

container
This parameter defines the containers to be run.

Required Container Parameters

Each *container* in the *service* contains at least the four following required parameters:

name
The name of the container.

image
The Docker image to use. If your image is in AWS ECR, you will use the full format:

```
<account number>.dkr.ecr.<region>.amazonaws.com/<image>:<version>
```

Since we're pulling an image from Dockerhub, we just need to supply the image name:

```
tutum/hello-world
```

cpu
The number of cpu units to reserve for the container.

memory
The hard limit of memory (in MB) available to the container.

3.1.4 Deploy

To deploy this service, add your configuration to the *deployfish.yml* file and in the same directory as your configuration file run:

```
deploy create hello-world-test
```

If you have named your configuration file something else, you can run:

```
deploy -f myconfigfile.yml create hello-world-test
```

Assuming everything ran successfully, you should be able to see the relevant info with:

```
deploy info hello-world-test
```

If you make a change and would like to update the service run:

```
deploy update hello-world-test
```

3.2 More Funtionality

3.2.1 Problem

In *A Basic Service*, we looked at the essentials of a service. We hosted an nginx based hello-world web site. A fundamental flaw with this site, though, is that it isn't accessible from anywhere but the local Docker container, which isn't terribly useful. We need to open the relevant ports to make it available. We're also going to set some environment variables and overwrite the Docker *command*.

3.2.2 Setup

We just need the same basic setup that we had in the first tutorial, namely an ECS cluster of at least one EC2 machine named *hello-world-cluster*

3.2.3 Configuration

Here's the configuration file for this service:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
        ports:
          - "80"
        command: /usr/bin/supervisord
```

(continues on next page)

(continued from previous page)

```
        environment:
          - VAR1=test
          - VAR2=anothervar
          - DEBUG=True
```

Here we've added three new parameters - *ports*, *command*, and *environment*:

ports
: This is a list of values, so each value begins with a dash. In our case, we are just opening up one port, so we just have the single value, *80*. This will open port 80, hosting it on a random port on the ECS cluster machine that is hosting the container.

command
: This is the Docker *command* that will be run when the container is started

environment
: This is a list of values, so each begins with a dash. It is always in the form:

  ```
  - VARIABLE=VALUE
  ```

  Anything set here will be available in the environment of the running container.

Port Options

If you want to specify the port number on the ECS cluster machine that will correspond to the container's port, you can specify that in the form HOST_PORT:CONTAINER_PORT:

```
ports:
  - "8000:80"
```

The *hello-world* web site will then be avialable on port 8000 of the ECS cluster machine that is hosting the container.

3.2.4 Deploy

To deploy this service, run the same command we ran in the last tutorial:

```
deploy create hello-world-test
```

3.3 Load Balancing

3.3.1 Problem

We often want to scale an application to run on more than one running container, either for performance or reliability reasons. In this tutorial, we'll add a load balancer to balance the load across two containers.

3.3.2 Setup

In addition to our basic setup from the previous tutorials, you need to create a load balancer. In this example, we're using an AWS Elastic Load Balancer (ELB) and naming it *hello-world-elb*.

3.3.3 Configuration

Here's the configuration file for this load balanced service:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    load_balancer:
      service_role_arn: arn:aws:iam::123445564666:role/ecsServiceRole
      load_balancer_name: hello-world-elb
      container_name: hello-world
      container_port: 80
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
        ports:
          - "80"
        command: /usr/bin/supervisord
        environment:
          - VAR1=test
          - VAR2=anothervar
          - DEBUG=True
```

Here we've added the new parameter, *load_balancer*. This corresponds to your AWS ELB.

Load Balancer Parameters

ELB

The *load_balancer* parameter requires the following four parameters if you are using a classic AWS ELB:

service_role_arn
: The name or full ARN of the IAM role that allows ECS to make calls to your load balancer on your behalf. You will need to use the ARN that corresponds to your account.

load_balancer_name
: The name of the ELB.

container_name
: The name of the container to associate with the load balancer

container_port
: The port on the container to associate with the load balancer. This port must correspond to a container port on container container_name in your service's task definition

ALB or NLB

AWS also offers the Application Load Balancers (ALB) and Network Load Balancers. If you are using one of those instead of the ELB, you will still use the *load_balancer* parameter, but it will require *target_group_arn* to be specified, rather than *load_balancer_name*:

target_group_arn
The full ARN of the target group to use for this service.

3.3.4 Deploy

To deploy this service, run the same command we ran in the last tutorial:

```
deploy create hello-world-test
```

To increase the number of running containers behind the load balancer to 2 instances, you can either modify the config, setting the count to:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 2
    family: hello-world
    load_balancer:
    ...
```

Then running *update*:

```
deploy update hello-world-test
```

Or you can scale the container arbitrarily with the *scale* command:

```
deploy scale test 2
```

3.4 Parameter Store

3.4.1 Problem

Most applications need some configuration. Some configuration can be passed as environment variables, but what about passwords and other secrets? Do you want them listed in the config file? These would then be visible to anyone who had access to your version control system. Any developer would also see all of them, including the production passwords. AWS introduced Parameter Store as part of Systems Manager. This allows us to store encrypted passwords and other secrets.

3.4.2 Setup

We'll start with the same setup as the initial tutorial, just an ECS cluster.

3.4.3 Configuration

Here's the configuration for this service:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
    config:
      - VAR1=value1
      - VAR2=value2
      - PASSWORD1:secure=password1
      - PASSWORD2:secure=password2
```

The new parameter here is *config*:

config
: This is a list of values, so each begins with a dash. For an unencrypted value, it is in the form:

  ```
  - VARIABLE=VALUE
  ```

  For an encrypted value, you must add the *secure* flag:

  ```
  - VARIABLE:secure=VALUE
  ```

  In this format, the encrypted value will be encrypted with the default key. For better security, make a unique key for each app and specify it in this format:

  ```
  - VARIABLE:secure:arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-
  ↪1234567890ab=VALUE
  ```

For more information about creating keys, see AWS Key Management Service (KMS).

Managing Config Parameters

In addition to deploying your services, you can also manage your config with *deployfish* using the *config* subcommand.

To view your current config in AWS, run:

```
deploy config show hello-world-test
```

To save config to AWS, run:

```
deploy config write hello-world-test
```

Reading From The Environment

You might have noticed that so far this solution is still displaying passwords in the *deployfish.yml* file for all the developers to see. This is not a good security practice as we've mentioned. The best way to deal with this is to have the secret parameter values defined in an environment variable. You would then change the *config* section to be:

```
...
config:
  - VAR1=value1
  - VAR2=value2
  - PASSWORD1:secure=${env.PASSWORD1}
  - PASSWORD2:secure=${env.PASSWORD2}
```

To make this easier, *deployfish* allows you to pass an environment file on the command line:

```
deploy --env_file=config.env create hello-world-test
```

This file is expected to be in the format:

```
VARIABLE=VALUE
VARIABLE=VALUE
```

These variables will all be loaded into the environment, so available to read from the *config* parameters. You would typically use a different file for each service.

You can also specify this file in the *service* definition itself:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    env_file: config.env
    ...
```

Using Config Parameters

So now that we have all of these values loaded into the AWS Parameter Store, how do we use them? We've included a subcommand in *deployfish* called *entrypoint*. You would define this as your *entrypoint* in your *Dockerfile*:

```
ENTRYPOINT ["deploy", "entrypoint"]
```

You would have to install *deployfish* in your container for this to work.

With this as your *entrypoint*, you will need to set the *command* parameter of the *container* to be your original *entrypoint*:

```
...
containers:
  - name: hello-world
    image: tutum/hello-world
    cpu: 128
    memory: 256
    command: /usr/bin/supervisord
...
```

The *entrypoint* that is run will then be:

```
deploy entrypoint <command>
```

or in this case:

```
deploy entrypoint /usr/bin/supervisord
```

When this is run, your defined *config* parameters will be downloaded from AWS Parameter Store and defined locally as environment variables, which you will then access as you would any environment variable.

If you run your docker container locally, the *entrypoint* subcommand will simply call the command without downloading anything from AWS Parameter Store. You would then use locally defined environment variables to set the various parameter values.

3.5 Using Terraform

3.5.1 Problem

If we use Terraform to build our infrastructure in AWS, we can use its outputs to populate the relevant portions of our `deployfish.yml` file.

3.5.2 Setup

We're going to presume a more sophisticated setup with an ECS cluster, an ELB, a task role to allow the container to have rights to other AWS services, an S3 bucket, and and RDS database.We'll also use the terraform state file that has been uploaded to S3.

3.5.3 Configuration

The Terraform Section

Here's the configuration file with terraform:

```
terraform:
  statefile: 's3://hello-world-remotestate-file/hello-world-terraform-state'
  lookups:
    cluster_name: 'test-cluster-name'
    load_balancer_name: 'test-elb-id'
    task_role_arn: 'iam-role-hello-world-test-task'
    rds_address: 'test-rds-address'
    app_bucket: 's3-hello-world-test-bucket'

services:
  - name: hello-world-test
    cluster: ${terraform.cluster_name}
    count: 1
    load_balancer:
      service_role_arn: arn:aws:iam::111122223333:role/ecsServiceRole
      load_balancer_name: ${terraform.load_balancer_name}
      container_name: hello-world
      container_port: 80
    family: hello-world
    task_role_arn: ${terraform.task_role_arn}
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
        ports:
          - "80"
        command: /usr/bin/supervisord
    config:
      - DB_NAME=hello_world
      - DB_USER=hello_world_u
      - DB_PASSWORD:secure=${env.DB_PASSWORD}
      - DB_HOST=${terraform.rds_address}
      - AWS_BUCKET=${terraform.app_bucket}
```

We first declare a `terraform:` section in the top-level of your `deployfish.yml` file. The values we define in that section are then available as a variable in `services:` section definitions, in the form `${terraform.variable_name}`. In the above config, we've defined `cluster` to be `${terraform.cluster_name}`. When we deploy, this will be automatically converted to:

```
cluster: test-cluster-name
```

Defining an Environment

We can take this a step further, though. Typically, we will use terraform to define all of the various environments, like test and prod. We can define the environment in our *service* definition with the *environment* parameter:

```
services:
  - name: hello-world-test
    cluster: ${terraform.cluster_name}
    environment: test
    count: 1
    ...
```

We can then use this environmant value in our `terraform:` section:

```
terraform:
  statefile: 's3://hello-world-remotestate-file/hello-world-terraform-state'
  lookups:
    cluster_name: '{environment}-cluster-name'
    load_balancer_name: '{environment}-elb-id'
    task_role_arn: 'iam-role-hello-world-{environment}-task'
    rds_address: '{environment}-rds-address'
    app_bucket: 's3-hello-world-{environment}-bucket'
...
```

Multiple Environments

This section can then be used for multiple service definitions under `services:` based on the different environments:

```
terraform:
  statefile: 's3://hello-world-remotestate-file/hello-world-terraform-state'
  lookups:
    cluster_name: '{environment}-cluster-name'
    load_balancer_name: '{environment}-elb-id'
    task_role_arn: 'iam-role-hello-world-{environment}-task'
    rds_address: '{environment}-rds-address'
    app_bucket: 's3-hello-world-{environment}-bucket'

services:
  - name: hello-world-test
    cluster: ${terraform.cluster_name}
    environment: test
    count: 1
    load_balancer:
      service_role_arn: arn:aws:iam::111122223333:role/ecsServiceRole
```

(continues on next page)

(continued from previous page)

```
    load_balancer_name: ${terraform.load_balancer_name}
    container_name: hello-world
    container_port: 80
  family: hello-world
  task_role_arn: ${terraform.task_role_arn}
  containers:
    - name: hello-world
      image: tutum/hello-world
      cpu: 128
      memory: 256
      ports:
        - "80"
      command: /usr/bin/supervisord
  config:
    - DB_NAME=hello_world
    - DB_USER=hello_world_u
    - DB_PASSWORD:secure=${env.DB_PASSWORD}
    - DB_HOST=${terraform.rds_address}
    - AWS_BUCKET=${terraform.app_bucket}

- name: hello-world-prod
  cluster: ${terraform.cluster_name}
  environment: prod
  count: 1
  load_balancer:
    service_role_arn: arn:aws:iam::111122223333:role/ecsServiceRole
    load_balancer_name: ${terraform.load_balancer_name}
    container_name: hello-world
    container_port: 80
  family: hello-world
  task_role_arn: ${terraform.task_role_arn}
  containers:
    - name: hello-world
      image: tutum/hello-world
      cpu: 256
      memory: 512
      ports:
        - "80"
      command: /usr/bin/supervisord
  config:
    - DB_NAME=hello_world
    - DB_USER=hello_world_u
    - DB_PASSWORD:secure=${env.DB_PASSWORD}
    - DB_HOST=${terraform.rds_address}
    - AWS_BUCKET=${terraform.app_bucket}
```

Here we defined both a *test* and *prod* environment. When we deploy *test* we will use one environment file to set the *config* parameters that contains the *test* values, and a *prod* environment file to define its values.

Another advantage of specifying an envieronment, is that you can use this environment in place of the service name when calling `deploy`.

Terraform List and Map Outputs

Terraform supports outputting lists and maps, and you can use lookups of list and map values in your service definitions:

```
terraform:
  statefile: 's3://hello-world-remotestate-file/hello-world-terraform-state'
  lookups:
    cluster_name: '{environment}-cluster-name'
    security_groups: 'service-security-groups'
    load_balancer: 'load-balancer-config'

services:
  - name: hello-world
    cluster: ${terraform.cluster_name}
    environment: prod
    count: 1
    load_balancer: ${terraform.load_balancer}
    vpc_configuration:
      security_groups: ${terraform.security_groups}
```

3.5.4 Deploy

To set the AWS Parameter Store values for *test*:

```
deploy --env_file=test.env config write test
```

Then for *prod*:

```
deploy --env_file=prod.env config write prod
```

The services are then created with:

```
deploy create test
```

and:

```
deploy create prod
```

3.6 Fargate Tutorial

3.6.1 Problem

In *More Funtionality*, we looked at an nginx based hello-world web site running on ECS EC2. In this tutorial we will see how to create the same service running on ECS Fargate.

3.6.2 Setup

We just need the same basic setup that we had in the first tutorial, namely an ECS cluster named *hello-world-cluster*, but we will not need any EC2 instances.

3.6.3 Configuration

Here's the configuration file for this service:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    network_mode: awsvpc
    launch_type: FARGATE
    execution_role: arn:aws:iam::123142123547:role/my-task-role
    cpu: 256
    memory: 512
    vpc_configuration:
      subnets:
        - subnet-12345678
        - subnet-87654321
      security_groups:
        - sg-12345678
      public_ip: ENABLED
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
        ports:
          - "80"
        command: /usr/bin/supervisord
        environment:
          - VAR1=test
          - VAR2=anothervar
          - DEBUG=True
```

You will notice that we have added several new parameters - *launch_type*, *execution_role*, *cpu*, *memory*, and *vpc_configuration*:

launch_type
: This is the parameter that specifies whether the service is an EC2 service or a FARGATE service. The default value is EC2 so you only need to specify this for a Fargate task.

execution_role
: This is the task exeuction role ARN for an IAM role that allows Fargate to pull container images and publish container logs to Amazon CloudWatch on your behalf

cpu
: For Fargate tasks you are required to define the cpu at the task level, and there are specific values that are allowed.

CPU value

256 (.25 vCPU) 512 (.5 vCPU) 1024 (1 vCPU) 2048 (2 vCPU) 4096 (4 vCPU)

memory
For Fargate tasks you are required to define the memory at the task level, and there are specific values that are allowed.

Memory value (MiB)

512 (0.5GB), 1024 (1GB), 2048 (2GB) 1024 (1GB), 2048 (2GB), 3072 (3GB), 4096 (4GB) 2048 (2GB), 3072 (3GB), 4096 (4GB), 5120 (5GB), 6144 (6GB), 7168 (7GB), 8192 (8GB) Between 4096 (4GB) and 16384 (16GB) in increments of 1024 (1GB) Between 8192 (8GB) and 30720 (30GB) in increments of 1024 (1GB)

vpc_configuration
The vpc configuration for any Fargate tasks requires the following four parameters:

subnets (array)
The subnets in the VPC that the task scheduler should consider for placement. Only private subnets are supported at this time. The VPC will be determined by the subnets you specify, so if you specify multiple subnets they must be in the same VPC.

security_groups (array)
The ID of the security group to associate with the service.

public_ip (string)
Whether to enabled or disable public IPs. Valid Values are `ENABLED` or `DISABLED`

3.6.4 Deploy

To deploy this service, run the same command we ran in the last tutorial:

```
deploy create hello-world-test
```

3.7 Advanced Features

3.7.1 Architectural Assumptions

A few assuptions are made as to how your VPCs are structured. It is assumed that you have a bastion host for each of your VPCs. These bastion hosts are used to access the individual EC2 instances in your ECS clusters. We expect these bastion hosts must also have a `Name` tag beginning with `bastion*`, like `bastion-test`, etc.

3.7.2 deploy cluster

The `deploy cluster` commands allow you to interract with the individual EC2 machines that make up your ECS cluster. It provides three subcommands, `info`, `run`, and `ssh`. For many of the advanced features of deployfish, the above assumptions have been made about your architecture that are required for them to work.

Info

The `info` subcommand allows you to view information about the individual EC2 systems that make up your ECS cluster. For example:

```
deploy cluster info web-test
```

Might return the output below:

```
Cluster: web-test
  pk                  :     web-test
  name                :     web-test
  arn                 :     arn:aws:ecs:us-west-2:123456789012:cluster/web-test
  status              :     ACTIVE
  instances           :     6
  autoscaling_group   :     web-test
  task counts
      running           :     5
      pending           :     0

Container instances
-------------------

Name               Instance Type    IP Address      Free CPU    Free Memory
-----------------  ---------------  ------------  ----------  -------------
ecs.web-test.b.2   t2.medium        10.0.1.1           768            102
ecs.web-test.b.1   t2.medium        10.0.1.2          1536            182
ecs.web-test.c.2   t2.medium        10.0.2.1          1408           1206
ecs.web-test.c.1   t2.medium        10.0.2.2          1024            614

Services
--------

Name                        Version      Desired    Running  Created
--------------------------  ---------  ---------  ---------  -------------------
service1                    2.0.8              2          2  2021-04-02 17:29:30
service2                    1.4.1              1          1  2021-04-23 11:21:39
service3                    2.1.2              2          2  2020-08-19 09:33:12
```

3.7.3 deploy service ssh <service_name>

The `deploy service ssh` command (alias: `deploy ssh`) will connect you via SSH to a system in your ECS cluster. If you have any running containers, it will choose one of those, otherwise it will connect to a random one. This is useful for debugging:

```
# ssh to a container instance for the service identified by environment "test' in␣
↪deployfish.yml
deploy service ssh test
# ssh to a container instance for the service "service1" in cluster "web-test"
deploy service ssh web-test:service1
```

3.7.4 deploy service exec <service_name>

The `deploy service exec` command (alias: `deploy exec`) will connect you to a running container, similar to connecting to the host running the container and running:

```
docker exec -it <contianer_id> /bin/bash
```

It will choose a random container. The command in our case would be:

```
# exec into a container for the service identified by environment "test' in deployfish.yml
deploy service exec test
# exec into a container for the service "service1" in cluster "web-test"
deploy service exec web-test:service1
```

CHAPTER

FOUR

DEPLOYFISH.YML REFERENCE

The deployfish service config file is a YAML file defining ECS services, task definitions and one-off tasks associated with those services.

- The default path for a deployfish configuration file is `./deployfish.yml`.
- If the environment variable `DEPLOYFISH_CONFIG_FILE` is defined, `deployfish` will use that instead.
- If you pass a filename to `deploy` with the `-f` or `--filename` command line flag, that will be used even if `DEPLOYFISH_CONFIG_FILE` is defined.

Options specified in the Dockerfile for your containers (e.g., `ENTRYPOINT`, `CMD`, `ENV`) are respected by default - you don't need to specify them again in `deployfish.yml`.

You can use terraform outputs in configuration values with a `${terraform.<key>}` syntax - see the *Interpolation* section for full details.

You can also use the values of environment variables in configuration values with a `${env.<key>}` syntax - see the *Interpolation* section for full details.

4.1 AWS Credentials

deployfish uses boto3 to do all its work in AWS and by default defers to boto3 credential resolution to figure out what AWS credentials it should use. See Configuring Credentials in boto3's documentation for details.

Alternately, you can tell deployfish specifically how to get your AWS credentials by defining an `aws:` section in `deployfish.yml`.

4.1.1 Static credentials

Static credentials can be provided by adding an `access_key` and `secret_key` in-line in an `aws:` section in `deployfish.yml`.

Usage:

```
aws:
  access_key: anaccesskey
  secret_key: asecretkey
  region: us-west-2
```

If you specify static credentials in this way, they will be used instead of any credentials found in your environment. `region` here is optional.

4.1.2 Using a profile from your AWS credentials file

You can use an AWS credentials file to specify your credentials and then set up your `aws:` section to use credentials from a particular profile. The default location is `$HOME/.aws/credentials` on Linux and OS X. You can specify a different location for this file via the `AWS_SHARED_CREDENTIALS_FILE` environment variable.

Usage:

```
aws:
  profile: customprofile
  region: us-west-2
```

`region` here is optional.

4.2 ECS Service Definition

This section contains a list of all configuration options supported by a ECS Service definition in version 1.

Services are specified in a YAML list under the top level `services:` key like so:

```
services:
  - name: foobar-prod
    ...
  - name: foobar-test
    ...
```

Unless otherwise specified, see Service Definition Parameters for help on thee options.

4.2.1 name

(String, Required) The name of the actual ECS service. `name` is required. The restrictions on characters in ECS services are in play here: Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

Once your service has been created, this is not changable without deleting and re-creating the service.

```
services:
  - name: foobar-prod
```

4.2.2 cluster

(String, Required) The name of the actual ECS cluster in which we'll create our service. `cluster` is required. This has to exist in AWS before running `deploy service create <service-name>`.

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
```

4.2.3 environment

(String, Optional) This is a keyword that can be used in terraform lookups (see "*Interpolation*", below). It can also be used as an alias for the service name in the `deploy` command.

```
services:
  - name: foobar-prod
    environment: prod
```

4.2.4 scheduling_strategy

(String, Optional) When we create the ECS service, configure the service to run in REPLICA or DAEMON. Default to REPLICA.

```
services:
  - name: foobar-prod
    clsuter: foodbar-cluster
    scheduling_strategy: DAEMON
```

See:

4.2.5 count

(Integer, Required for REPLICA scheduling strategy) When we create the ECS service, configure the service to run this many tasks.

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
```

`count` is only meaningful at service creation time. To change the count in an already created service, use `deploy service scale <service_name> <count>`

4.2.6 maximum_percent

(Integer, Optional) During a deployment, this is the upper limit on the number of tasks that are allowed in the RUNNING or PENDING state, as a percentage of the `count`. This must be configured along with `minimum_healthy_percent`. If not provided will default to 200. If schdeuling strategy is set to DAMEON, it will be fixd at 100

```
services:
  - name: foobar-prod
    maximum_percent: 200
```

4.2.7 minimum_healthy_percent

(Integer, Optional) During a deployment,this is the lower limit on the number of tasks that must remain in the RUNNING state, as a percentage of the `count`. This must be configured along with `maximum_percent`. If not provided will default to 0.

```
services:
  - name: foobar-prod
    minimum_healthy_percent: 50
```

4.2.8 placement_constraints

(Optional) An array of placement constraint objects to use for tasks in your service. You can specify a maximum of 10 constraints per task (this limit includes constraints in the task definition and those specified at run time).

```
services:
    - name: foobar-prod
      placement_constraints:
        - type: distinctInstance
        - type: memberOf
          expression: 'attribute:ecs.instance-type =~ t2.*'
```

4.2.9 placement_strategy

(Optional) The placement strategy objects to use for tasks in your service. You can specify a maximum of four strategy rules per service.

```
services:
    - name: foobar-prod
      placement_strategy:
        - type: random
        - type: spread
          field: 'attribute:ecs.availability-zone'
```

See Service Definition Parameters.

4.2.10 launch_type

The launch type on which to run your service. Accepted values are `FARGATE` or `EC2`. If a launch type is not specified, `EC2` is used by default.

If you use the Fargate launch type, these task parameters are not valid:

- `dockerSecurityOptions`
- `links`
- `linuxParameters`
- `placementConstraints`
- `privileged`

Example:

```
services:
  - name: foobar-prod
    launch_type: FARGATE
```

See Amazon ECS Launch Types.

4.2.11 enable_exec

If "`true`", enable ECS Exec for the tasks on this service. If `enable_exec` is not specified, default to ""`false`"".

Important: In addition to setting this to "`true`", in order for ECS Exec to work, you'll need to configure your cluster, task role and the system on which you run deployfish as described here: Using Amazon ECS Exec for debugging.

4.2.12 vpc_configuration

If you are configuring a `FARGATE` task or you have tasks with the `awsvpc` network mode, you must specify your vpc configuration at the task level.

deployfish won't create the VPC, subnets or security groups for you – you'll need to create it before you can use `deploy service create <service_name>`

You'll need to specify

- `subnets`: (list of strings) The subnets in the VPC that the task scheduler should consider for placement. Only private subnets are supported at this time. The VPC will be determined by the subnets you specify, so if you specify multiple subnets they must be in the same VPC.
- `security_groups`: (list of strings) The ID of the security group to associate with the service.
- `public_ip`: (string) Whether to enabled or disable public IPs. Valid values are `ENABLED` or `DISABLED`

Example:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    vpc_configuration:
      subnets:
        - subnet-12345678
        - subnet-87654321
      security_groups:
        - sg-12345678
      public_ip: DISABLED
```

4.2.13 autoscalinggroup_name

(Optional)

If you have a dedicated EC2 AutoScaling Group for your service, you can declare it with the `autoscalinggroup_name` option. This will allow you to scale the ASG up and down when you scale the service up and down with `deploy service scale <service-name> <count>`.

deployfish won't create the autoscaling group for you – you'll need to create it before you can use `deploy service scale <service_name> <count>` to manipulate it.

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    autoscalinggroup_name: foobar-asg
```

Alternatively, you can specify an AutoScaling Group Capacity Provider for this service, and the scaling will be taken care of automatically.

4.2.14 volumes

(Optional)

You can define volumes that can be mounted inside your task's containers via the `volumes` section of your deployfish service definition. You only really need to do use this if you want to use a docker volume driver that is not the built in `local` one – the one that allows you to mount host machinefolders into your container. To mount one of the volumes you define here in one of your containers, see "volumes" under "Container Definitions" on this page.

Here is a fully specified example

```
services:
  - name: foobar-prod
    cluster: foobar-prod
    volumes:
      - name: storage_task
        config:
          scope: task
          autoprovision: true
          driver: my_vol_driver:latest
      - name: storage_shared
        config:
          scope: shared
          driver: my_vol_driver:latest
          driverOpts:
            opt1: value1
            opt2: value2
          labels:
            key: value
            key: value
      - name: efs_storage
        efs_config:
          file_system_id: my-file-system-id
          root_directory: my-root-directory
```

(continues on next page)

(continued from previous page)

```
    - name: local_storage
      path: /host/path
```

The above defines four volumes:

- (EC2 launch type only) a task specific (not usable by other tasks) volume named `storage_task` that will be autocreated and which will use the `my_vol_driver:latest` volume driver
- (EC2 launch type only) a shared (usable by other tasks) volume named `storage` that uses the docker volume driver `my_vol_driver:latest` with the driver options given in the `driverOpts:` section (driver options are volume driver specific) and labels given by `labels`.
- (Both EC2 or FARGATE launch types) a volume named `efs_storage` that allows you is the EFS file system `my-filesystem-id`, rooted in the folder `my-root-directory`. Note: *root_directory* is optional, and if ommitted will be set to `/`.
- (Both EC2 or FARGATE launch types) a volume named `local_storage` that just allows you to mount `/host/path` from the host machine using the builtin `local` volume driver. For this type of mount, you can also mount `/host/path` directly via the `volumes` section of your container definition and not define it here.

See Using Data Volumes in Tasks.

Note: You are responsible for installing and confuring any 3rd party docker volume drivers on your ECS container machines. The *volumes* section just allows you to use that driver once you've properly set it up and configured it.

4.2.15 service_role_arn

(Optional)

Note: You should only specify `service_role_arn` if you do not have the `AWSServiceRoleForECS``a service linked role in your account and you are not using ``awsvpc` network mode on your task definition. If you do have that role, ECS will use it automatically and will not allow you to create your service until you remove `service_role_arn`.

The name or full Amazon Resource Name (ARN) of the IAM role that allows Amazon ECS to make calls to your load balancer on your behalf. This parameter is only permitted if you are using a load balancer with your service and your task definition does not use the `awsvpc` network mode. If you specify the role parameter, you must also specify a load balancer object with the `load_balancer` parameter, below.

Example:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    service_role_arn: arn:aws:iam::123142123547:role/ecsServiceRole
    load_balancer:
      load_balancer_name: foobar-prod-elb
      container_name: foobar-prod
      container_port: 80
```

See: Using Service-Linked ROles for Amazon ECS

4.2.16 load_balancer

(Optional)

If you're going to use an ELB or an ALB with your service, configure it with a `load_balancer` block.

The load balancer info for the service can't be changed after the service has been created. To change any part of the load balancer info, you'll need to destroy and recreate the service.

See: Service Load Balancing.

ELB

To specify that the the service is to use an ELB, you'll need to specify

- `load_balancer_name`: (string) The name of the ELB.
- `container_name`: (string) the name of the container to associate with the load balancer
- `container_port`: (string) the port on the container to associate with the load balancer. This port must correspond to a container port on container `container_name` in your service's task definition

Example:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    service_role_arn: arn:aws:iam::123142123547:role/ecsServiceRole
    load_balancer:
      load_balancer_name: foobar-prod-elb
      container_name: foobar-prod
      container_port: 80
```

deployfish won't create the load balancer for you – you'll need to create it before running `deploy service create <service_name>`.

ALB or NLB

To specify that the the service is to use an ALB or NLB, you'll need to specify:

- `target_group_arn`: (string) The full ARN of the target group to use for this service.
- `container_name`: (string) the name of the container to associate with the load balancer
- `container_port`: (string) the port on the container to associate with the load balancer. This port must correspond to a container port on container `container_name` in your service's task definition

---

Note: If you set `network_mode` to `awsvpc` or you've set `launch_type` to `FARGATE`, you need to configure your ALB/NLB target group to target IP addresses, not EC2 instances. This is because tasks that use the awsvpc network mode are associated with an elastic network interface, not an Amazon EC2 instance.

See: Service Load Balancing

---

deployfish won't create the target group for you == you'll need to create it before running `deploy service create <service_name>`.

Example:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    service_role_arn: arn:aws:iam::123142123547:role/ecsServiceRole
    load_balancer:
      target_group_arn: my-target-group-arn
      container_name: foobar-prod
      container_port: 80
```

You can specify multiple target groups for your service, by placing them in a list named `target_groups`:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    service_role_arn: arn:aws:iam::123142123547:role/ecsServiceRole
    load_balancer:
      target_groups:
      - target_group_arn: my-target-group-arn-80
        container_name: foobar-prod
        container_port: 80
      - target_group_arn: my-target-group-arn-443
        container_name: foobar-prod
        container_port: 443
```

See: Registering Multiple Target Groups with a Service

4.2.17 capacity_provider_strategy

(Optional)

Define a list of one or more capacity providers with weights for this service. Capacity providers allow the service to control the underlying Fargate cluster or AutoScaling Group to allocate more container machines when necessary to support your service requirements. Any capacity provider you name in your strategies must already be associated with the cluster.

Note: `capacity_provider_strategy` and `launch_type` are mutually exclusive. Define one or the other. To use Fargate with *capacity_provider_strategy*, choose either the `FARGATE` or `FARGATE_SPOT` pre-defined providers.

Example

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    capacity_provider_strategy:
    - provider: foobar-cap-provider
      weight: 1
      base 1
    - provider: foobar-cap-provider-spot
      weight: 2
```

See the description of the `capacityProviderStrategy` parameter in the boto3 ECS create_service() documentation.

4.2.18 service_discovery

(Optional)

If you're going to use ECS service discovery, configure it with a `service_discovery` block.

The service discovery info for the service can't be changed after the service has been created. To change any part of the service discovery info, you'll need to destroy and recreate the service.

To use service discovery you'll need to specify

- `namespace`: (string) The service discovery namespace that the new service will be associated with.
- `name`: (string) The name of the service discovery service
- **`dns_records`: (list) A list of DNS records the service discovery service should create**
  - `type`: (string) The type of dns record. Valid values are `A` and `SRV`.
  - `ttl`: (int) The ttl of the dns record.

Example:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    service_discovery:
      namespace: local
      name: foobar-prod
      dns_records:
        type: A
        ttl: 10
```

This would create a new service discovery service on the `local` Route53 private zone. The DNS would be `foobar-prod.local`

See Amazon ECS Service Discovery.

4.2.19 application_scaling

(Optional)

If you want your service so scale up and down with service CPU, configure it with an `application_scaling` block.

Example:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    application_scaling:
        min_capacity: 2
        max_capacity: 4
        role_arn: arn:aws:iam::123445678901:role/ApplicationAutoscalingECSRole
        scale-up:
```

(continues on next page)

(continued from previous page)

```
            cpu: ">=60"
            check_every_seconds: 60
            periods: 5
            cooldown: 60
            scale_by: 1
        scale-down:
            cpu: "<=30"
            check_every_seconds: 60
            periods: 30
            cooldown: 60
            scale_by: -1
```

This block says that, for this service:

- There should be a minimum of 2 tasks and a maximum of 4 tasks * `arn:aws:iam::123445678901:role/ApplicationAutoscalingECSRole` grants permission to start new containers for our service
- Scale our service up by one task if ECS Service Average CPU is greater than 60 percent for 300 seconds. Don't scale up more than once every 60 seconds.
- Scale our service down by one task if ECS Service Average CPU is less than or equal to 30 percent for 1800 seconds. Don't scale down more than once every 60 seconds.

min_capacity

(Integer, Required) The minimum number of tasks that should be running in our service.

max_capacity

(Integer, Required) The maximum number of tasks that should be running in our service. Note that you should ensure that you have enough resources in your cluster to actually run this many of your tasks.

role_arn

(String, Required) The name or full ARN of the IAM role that allows Application Autoscaling to muck with your service. Your role definition should look like this:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "application-autoscaling.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

And it needs an appropriate policy attached. The below policy allows the role to act on any service.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1456535218000",
            "Effect": "Allow",
            "Action": [
                "ecs:DescribeServices",
                "ecs:UpdateService"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "Stmt1456535243000",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:DescribeAlarms"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

See Amazon ECS Service Auto Scaling IAM Role.

scale-up, scale-down

(Required) You should have exactly two scaling rules sections, and they should be named precisely `scale-up` and `scale-down`.

cpu

(String, Required) What CPU change causes this rule to be activated? Valid operators are: <=, <, >, >=. The CPU value itself is a float.

You'll need to put quotes around your value of `cpu`, else the YAML parser will freak out about the = sign.

check_every_seconds

(Integer, Required) Check the Average service CPU every this many seconds.

periods

(Integer, Required) The `cpu` test must be true for `check_every_seconds * periods` seconds for scaling to actually happen.

scale_by

(Integer, Required) When it's time to scale, scale by this number of tasks. To scale up, make the number positive; to scale down, make it negative.

cooldown

(Integer, Required) The amount of time, in seconds, after a scaling activity completes where previous trigger-related scaling activities can influence future scaling events.

See "Cooldown" in AWS' PutScalingPolicy documentation.

4.2.20 family

(String, Required) When we create task definitions for this service, put them in this family. When you go to the "Task Definitions" page in the AWS web console, what is listed under "Task Definition" is the family name.

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
```

See also the AWS documentation.

4.2.21 network_mode

(String, Optional) The Docker networking mode for the containers in our task. One of: `bridge`, `host`, `awsvpc` or `none`. If this parameter is omitted, a service is assumed to use `bridge` mode.

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: bridge
```

See the AWS documentation for what each of those modes are.

In order to be able to specify `awsvpc` as your network mode, you also need to define `vpc_configuration`:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: awsvpc
```

(continues on next page)

(continued from previous page)

```
    vpc_configuration:
      subnets:
        - subnet-12345678
        - subnet-87654321
      security_groups:
        - sg-12345678
      public_ip: DISABLED
```

4.2.22 task_role_arn

(String, Optional) A task role ARN for an IAM role that allows the containers in the task permission to call the AWS APIs that are specified in its associated policies on your behalf.

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: bridge
    task_role_arn: arn:aws:iam::123142123547:role/my-task-role
```

deployfish won't create the Task Role for you – you'll need to create it before running `deploy service create <service_name>`.

See also the AWS documentation, and IAM Roles For Tasks

4.2.23 execution_role

(String, Required for Fargate) A task exeuction role ARN for an IAM role that allows Fargate to pull container images and publish container logs to Amazon CloudWatch on your behalf.:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: bridge
    execution_role: arn:aws:iam::123142123547:role/my-task-role
```

deployfish won't create the Task Execution Role for you – you'll need to create it before running `deploy service create <service_name>`.

See also the IAM Roles For Tasks

4.2.24 cpu

(Required for Fargate tasks)

If you are configuring a Fargate task, you have to specify the cpu at the task level, and there are specific values for cpu which are supported which we describe below.

The available CPU values are:

Value	Virtual CPUs
256	.25 vCPU
512	.5 vCPU
1024	1 vCPU
2048	2 vCPU
4096	4 vCPU

See also the Task Definition Parameters

4.2.25 memory

(Required for Fargate tasks)

If you are configuring a Fargate task, you have to specify the memory at the task level, and there are specific values for memory which are supported which we describe below.

The available memory choices for a specific CPU value are:

CPU	Memory Configurations
256 (.25 vCPU)	512 (0.5GB), 1024 (1GB), 2048 (2GB)
512 (.5 vCPU)	1024 (1GB), 2048 (2GB), 3072 (3GB), 4096 (4GB)
1024 (1 vCPU)	2048 (2GB), 3072 (3GB), 4096 (4GB), 5120 (5GB), 6144 (6GB), 7168 (7GB), 8192 (8GB)
2048 (2 vCPU)	Between 4096 (4GB) and 16384 (16GB) in increments of 1024 (1GB)
4096 (4 vCPU)	Between 8192 (8GB) and 30720 (30GB) in increments of 1024 (1GB)

See also the Task Definition Parameters

4.3 ECS Task Configuration

This section contains a list of all configuration options supported by a ECS Task definition in version 1.

Services are specified in a YAML list under the top level `tasks:` key like so:

```
tasks:
  - name: foobar-prod
    ...
  - name: foobar-test
    ...
```

4.3.1 name

(String, Required) The name of the actual ECS tasks. `name` is required. The restrictions on characters in ECS tasks are in play here: Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

tasks:

- name: foobar-prod

4.3.2 service

(String, Option) Use the `service` option to associate this task with a particular service. This is used when running `deploy service service tasks <service_name>`.

tasks:

- name: foobar-prod service: foobar-service-prod

4.3.3 cluster

(String, Required) The name of the actual ECS cluster in which we'll run our task.:

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
```

4.3.4 environment

(String, Optional) This is a keyword that can be used in terraform lookups (see "*Interpolation*", below). It can also be used as an alias for the task name in the `deploy` command.

```
tasks:
  - name: foobar-prod
    environment: prod
```

4.3.5 count

(Integer) When we run the ECS task, run this many instances.

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
```

4.3.6 launch_type

(Required for Fargate tasks)

If you are configuring a Fargate task you must specify the launch type as `FARGATE`, otherwise the default value of `EC2` is used.

The Fargate launch type allows you to run your containerized applications without the need to provision and manage the backend infrastructure. Just register your task definition and Fargate launches the container for you.

If you use the Fargate launch type, the following task parameters are not valid:

- `dockerSecurityOptions`
- `links`
- `linuxParameters`
- `placementConstraints`
- `privileged`

Example:

```
tasks:
  - name: foobar-prod
    launch_type: FARGATE
```

See Amazon ECS Launch Types.

4.3.7 vpc_configuration

(Required for Fargate tasks)

If you are configuring a Fargate task, you have to specify your vpc configuration at the task level.

deployfish won't create the vpc, subnets or security groups for you – you'll need to create it before you can use `deploy task run <task_name>`

You'll specify

- `subnets`: (array) REQUIRED The subnets in the VPC that the task scheduler should consider for placement. Only private subnets are supported at this time. The VPC will be determined by the subnets you specify, so if you specify multiple subnets they must be in the same VPC.
- `security_groups`: (array) OPTIONAL The ID of the security group to associate with the task.
- `public_ip`: (string) OPTIONAL Whether to enabled or disable public IPs. Valid Values are `ENABLED` or `DISABLED`

Example:

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    launch_type: FARGATE
    vpc_configuration:
      subnets:
        - subnet-12345678
        - subnet-87654321
```

(continues on next page)

(continued from previous page)

```
      security_groups:
        - sg-12345678
      public_ip: ENABLED
```

4.3.8 volumes

(Optional)

You can define volumes that can be mounted inside your task's containers via the `volumes` section of your deployfish task definition. You only really need to do use this if you want to use a docker volume driver that is not the built in `local` one – the one that allows you to mount host machinefolders into your container. To mount one of the volumes you define here in one of your containers, see "volumes" under "Container Definitions" on this page.

Here is a fully qualfied example

```
tasks:
  - name: foobar-prod
    cluster: foobar-prod
    volumes:
      - name: storage_task
        config:
          scope: task
          autoprovision: true
          driver: my_vol_driver:latest
      - name: storage
        config:
          scope: shared
          driver: my_vol_driver:latest
          driverOpts:
            opt1: value1
            opt2: value2
          labels:
            key: value
            key: value
      - name: local_storage
        path: /host/path
```

The above defines three volumes:

- (EC2 launch type only) a task specific (not usable by other tasks) volume named `storage_task` that will be autocreated and which will use the `my_vol_driver:latest` volume driver
- (EC2 launch type only) a shared (usable by other tasks) volume named `storage` that uses the docker volume driver `my_vol_driver:latest` with the driver options given in the `driverOpts:` section (driver options are volume driver specific) and labels given by `labels`.
- (Both EC2 or FARGATE launch types) a volume named `local_storage` that just allows you to mount `/host/path` from the host machine using the builtin `local` volume driver. For this type of mount, you can also mount `/host/path` directly via the `volumes` section of your container definition and not define it here.

See Using Data Volumes in Tasks.

Note: You are responsible for installing and confuring any 3rd party docker volume drivers on your ECS container

machines. The *volumes* section just allows you to use that driver once you've properly set it up and configured it.

4.3.9 family

(String, Required) When we create task definitions for this task, put them in this family. When you go to the "Task Definitions" page in the AWS web console, what is listed under "Task Definition" is the family name.

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
```

See also the AWS documentation.

4.3.10 network_mode

(String, Optional) The Docker networking mode for the containers in our task. One of: `bridge`, `host`, `awsvpc` or `none`. If this parameter is omitted, a task is assumed to use `bridge` mode.

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: bridge
```

See the AWS documentation for what each of those modes are.

4.3.11 task_role_arn

(String, Optional) A task role ARN for an IAM role that allows the containers in the task permission to call the AWS APIs that are specified in its associated policies on your behalf.

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: bridge
    task_role_arn: arn:aws:iam::123142123547:role/my-task-role
```

deployfish won't create the Task Role for you – you'll need to create it before running `deploy task run <task_name>`.

See also the AWS documentation, and IAM Roles For Tasks

4.3.12 execution_role

(String, Required for Fargate) A task exeuction role ARN for an IAM role that allows Fargate to pull container images and publish container logs to Amazon CloudWatch on your behalf.:

```
tasks:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    family: foobar-prod-task-def
    network_mode: bridge
    execution_role: arn:aws:iam::123142123547:role/my-task-role
```

deployfish won't create the Task Execution Role for you – you'll need to create it before running `deploy task run <task_name>`.

See also the IAM Roles For Tasks

4.3.13 cpu

(Required for Fargate tasks)

If you are configuring a Fargate task, you have to specify the cpu at the task level, and there are specific values for cpu which are supported which we describe below.

The available CPU values are:

Value	Virtual CPUs
256	.25 vCPU
512	.5 vCPU
1024	1 vCPU
2048	2 vCPU
4096	4 vCPU

See also the Task Definition Parameters

4.3.14 memory

(Required for Fargate tasks)

If you are configuring a Fargate task, you have to specify the memory at the task level, and there are specific values for memory which are supported which we describe below.

The available memory choices for a specific CPU value are:

CPU	Memory Configurations
256 (.25 vCPU)	512 (0.5GB), 1024 (1GB), 2048 (2GB)
512 (.5 vCPU)	1024 (1GB), 2048 (2GB), 3072 (3GB), 4096 (4GB)
1024 (1 vCPU)	2048 (2GB), 3072 (3GB), 4096 (4GB), 5120 (5GB), 6144 (6GB), 7168 (7GB), 8192 (8GB)
2048 (2 vCPU)	Between 4096 (4GB) and 16384 (16GB) in increments of 1024 (1GB)
4096 (4 vCPU)	Between 8192 (8GB) and 30720 (30GB) in increments of 1024 (1GB)

See also the Task Definition Parameters

4.3.15 placement_constraints

(Optional) An array of placement constraint objects to use for tasks. You can specify a maximum of 10 constraints per task (this limit includes constraints in the task definition and those specified at run time).

```
tasks:
    - name: foobar-prod
      placement_constraints:
        - type: distinctInstance
        - type: memberOf
          expression: 'attribute:ecs.instance-type =~ t2.*'
```

See Task Placement Constraints.

4.3.16 placement_strategy

(Optional) The placement strategy objects to use for tasks in your service. You can specify a maximum of four strategy rules per service.

```
services:
    - name: foobar-prod
      placement_strategy:
        - type: random
        - type: spread
          field: 'attribute:ecs.availability-zone'
```

See Task Placement Strategies.

4.3.17 platform_version

(Optional) The platform version the task should run. A platform version is only specified for tasks using the Fargate launch type. If one is not specified, the LATEST platform version is used by default.

See AWS Fargate Platform Versions.

4.3.18 group

The name of the task group to associate with the task. The default value is the family name of the task definition.

4.3.19 schedule

The scheduling expression. For example, "cron(0 20 * * ? *)" or "rate(5 minutes)".

See Schedule Expressions for Rules.

4.3.20 schedule_role

The Amazon Resource Name (ARN) of the IAM role associated with the schedule rule. This should just allow the cloudwatch scheduled event to run the task. It should have a policy like:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "*"
        },
        {
            "Sid": "Stmt1455323356000",
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

4.4 Container Definitions

Define your containers within a task or service by using a `containers:` subsection.

`containers` is a list of containers like so:

```
services:
  - name: foobar-prod
    cluster: foobar-cluster
    count: 2
    containers:
      - name: foo
        image: my_repository/foo:0.0.1
        cpu: 128
        memory: 256
      - name: bar
        image: my_repository/baz:0.0.1
        cpu: 256
        memory: 1024
```

Each of the containers listed in the `containers` list will be added to the task definition for the service.

For each of the following attributes, see also the AWS ECS documentation.

NOTE: Each container in your service automatically gets their log configuration setup as 'fluentd', with logs being sent to `127.0.0.1:24224` and being tagged with the name of the container.

4.4.1 name

(String, Required) The name of the container. If you are linking multiple containers together in a task definition, the name of one container can be entered in the links of another container to connect the containers. The restrictions on characters in ECS container are in play here: Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

```
containers:
  - name: foo
```

4.4.2 image

(String, Required) The image used to start the container. Up to 255 letters (uppercase and lowercase), numbers, hyphens, underscores, colons, periods, forward slashes, and number signs are allowed.

For an AWS ECR repository:

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
```

For a Docker hub repository:

```
containers:
  - name: foo
    image: centos:7
```

4.4.3 memory

(Integer, Required) The hard limit of memory (in MB) available to the container. If the container tries to exceed this amount of memory, it is killed.

```
containers:
  - name: foo
    image: centos:7
    memory: 512
```

4.4.4 memoryReservation

(Integer, Optional) The soft limit (in MB) of memory to reserve for the container. When system memory is under heavy contention, Docker attempts to keep the container memory to this soft limit; however, your container can consume more memory when it needs to, up to the hard limit specified with the `memory` parameter. `memoryReservation` must be less than `memory`

```
containers:
  - name: foo
    image: centos:7
    memory: 512
    memoryReservation: 256
```

For example, if your container normally uses 128 MiB of memory, but occasionally bursts to 256 MiB of memory for short periods of time, you can set a memoryReservation of 128 MiB, and a memory hard limit of 300 MiB. This configuration would allow the container to only reserve 128 MiB of memory from the remaining resources on the container instance, but also allow the container to consume more memory resources when needed.

4.4.5 cpu

(Integer, Required) The number of cpu units to reserve for the container. A container instance has 1,024 cpu units for every CPU core.

```
containers:
  - name: foo
    image: centos:7
    cpu: 128
```

4.4.6 ports

(List of strings, Optional) A list of port mappings for the container.

Either specify both ports (HOST:CONTAINER), or just the container port (a random host port will be chosen). You can also specify a protocol as (HOST:CONTAINER/PROTOCOL). Note that both HOST and CONTAINER here must be single ports, not port ranges as `docker-compose.yml` allows in its port definitions. PROTOCOL must be one of 'tcp' or 'udp'. If no PROTOCOL is specified, we assume 'tcp'.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    ports:
    - "80"
    - "8443:443"
    - "8125:8125/udp"
```

4.4.7 links

(List of strings, Optional) A list of names of other containers in our task definition. Adding a container name to links allows containers to communicate with each other without the need for port mappings.

Links should be specified as `CONTAINER_NAME`, or `CONTAINER_NAME:ALIAS`.

```
containers:
  - name: my-service
    image: 123445564666.dkr.ecr.us-west-2.amazonaws.com/my-service:0.1.0
    cpu: 128
    memory: 256
    links:
      - redis
      - db:database
  - name: redis
    image: redis:latest
    cpu: 128
    memory: 256
```

(continues on next page)

(continued from previous page)

```
  - name: db
    image: mysql:5.5.57
    cpu: 128
    memory: 512
    environment:
        MYSQL_ROOT_PASSWORD: __MYSQL_ROOT_PASSWD__
```

4.4.8 essential

(Boolean, Optional) If the essential parameter of a container is marked as true, and that container fails or stops for any reason, all other containers that are part of the task are stopped. If the essential parameter of a container is marked as false, then its failure does not affect the rest of the containers in a task. If this parameter is omitted, a container is assumed to be essential.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    essential: true
  - name: bar
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    essential: false
```

4.4.9 extra_hosts

(list of strings, Optional) Add hostname mappings.

```
containers:
  - name: foo
    extra_hosts:
    - "somehost:162.242.195.82"
    - "otherhost:50.31.209.229"
```

An entry with the ip address and hostname will be created in `/etc/hosts` inside containers for this service, e.g:

```
162.242.195.82  somehost
50.31.209.229   otherhost
```

4.4.10 entrypoint

(String, Optional) The entry point that is passed to the container. Specify it as a string and Deployintaor will split the string into an array for you for passing to ECS.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    entrypoint: /entrypoint.sh here are arguments
```

4.4.11 command

(String, Optional) The command that is passed to the container. Specify it as a string and Deployintaor will split the string into an array for you for passing to ECS.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    command: apachectl -DFOREGROUND
```

4.4.12 environment

(Optional) Add environment variables. You can use either an array or a dictionary. Any boolean values; true, false, yes no, need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    environment:
      DEBUG: 'True'
      ENVIRONMENT: prod
      SECERTS_BUCKET_NAME: my-secrets-bucket
  - name: bar
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    environment:
      - DEBUG=True
      - ENVIRONMENT=prod
      - SECERTS_BUCKET_NAME=my-secrets-bucket
```

4.4.13 ulimits

(Optional) Override the default ulimits for a container. You can either specify a single limit as an integer or soft/hard limits as a mapping.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    ulimits:
      nproc: 65535
      nofile:
        soft: 65535
        hard: 65535
```

See Task Definition Parameters: Resource Limits.

4.4.14 cap_add

(List of strings, Optional) List here any Linux kernel capabilities your container should have

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    cap_add:
      - SYS_ADMIN
      - CHOWN
```

Note: The capabilities should be in ALL CAPS. Valid values are given in the link below.

See Task Definition Parameters: Linux Parameters.

4.4.15 cap_drop

(List of strings, Optional) List here any Linux kernel capabilities your container should **not** have

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    cap_drop:
      - SYS_RAWIO
```

Note: The capabilities should be in ALL CAPS. Valid values are given in the link below.

4.4.16 tmpfs

(Optional) The container path, mount options, and size (in MiB) of the tmpfs mount. This parameter maps to the –tmpfs option to docker run, mount_options is optional

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    tmpfs:
      - container_path: /tmpfs
        size: 256
        mount_options:
          - defaults
          - noatime
      - container_path: /tmpfs_another
        size: 128
```

See Task Definition Parameters: Linux Parameters.

4.4.17 dockerLabels

(Optional) Add metadata to containers using Docker labels. You can use either an array or a dictionary.

Use reverse-DNS notation to prevent your labels from conflicting with those used by other software.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    dockerLabels:
    labels:
      edu.caltech.description: "Fun webapp"
      edu.caltech.department: "Dept. of Redundancy Dept."
      edu.caltech.label-with-empty-value: ""
  - name: bar
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    dockerLabels:
      - "edu.caltech.description=Fun webapp"
      - "edu.caltech.department=Dept. of Redundancy Dept."
      - "edu.caltech.label-with-empty-value"
```

4.4.18 volumes

(List of strings, Optional) Specify a path on the host machine (VOLUME:CONTAINER), or an access mode (VOLUME:CONTAINER:ro). The HOST and CONTAINER paths should be absolute paths.

```
containers:
  - name: foo
    image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
    volumes:
      - /host/path:/container/path
      - /host/path-ro:/container/path-ro:ro
```

If you set the VOLUME portion of the mount to a filesystem path (e.g. "`/host/path`" in the above example), deployfish will mount that folder on the host machine into your container via the *local* docker volume driver. You won't need to define the volume specifically in the `volumes` section in your task definition.

You can also set the VOLUME portion of the mount to the name of a volume defined in your task definition's `volumes` section

```
services:
  - name: foobar
    cluster: foobar
    containers:
      - name: foo
        image: 123142123547.dkr.ecr.us-west-2.amazonaws.com/foo:0.0.1
        volumes:
          - storage:/container/path
    volumes:
      - name: storage
        config:
          scope: shared
          driver: rexray/s3fs:0.11.1
```

The above will cause the volume named `storage` from the docker volume driver `rexray/s3fs:0.11.1` to be mounted inside your container on `/container/path`

4.4.19 logging

(String and dictionary, Optional) Specify a log driver and its associated options.

To configure awslogs:

```
logging:
  driver: awslogs
  options:
    awslogs-group: awslogs-mysql
    awslogs-region: ap-northeast-1
    awslogs-stream-prefix: awslogs-example
```

For fluentd:

```
logging:
  driver: fluentd
  options:
    fluentd-address: 127.0.0.1:24224
    tag: hello
```

NOTE: if you don't provide a `logging:` section, no logs will be emitted from your service.

4.5 Secrets Management with AWS Parameter Store

The `config:` subsection of an ECS service or task is a list of parameters that are stored in the AWS Parameter Store as part of Systems Manager. This allows us to store settings, encrypted passwords and other secrets without exposing them to casual view in the AWS Console via the `environment` section of the container definition.

This is a list, so each item begins with a dash. For an unencrypted value, it is in the form:

```
- VARIABLE=VALUE
```

For an encrypted value, you must add the *secure* flag:

```
- VARIABLE:secure=VALUE
```

In this format, the encrypted value will be encrypted with the default key. For better security, make a unique key for each app and specify it in this format:

```
- VARIABLE:secure:arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-
↪1234567890ab=VALUE
```

For more information about creating keys, see AWS Key Management Service (KMS).

Here's an example configuration:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
```

(continues on next page)

(continued from previous page)

```
    count: 1
    family: hello-world
    containers:
      - name: hello-world
        image: tutum/hello-world
        cpu: 128
        memory: 256
    config:
      - VAR1=value1
      - VAR2=value2
      - PASSWORD1:secure=password1
      - PASSWORD2:secure=password2
```

4.5.1 Managing Config Parameters in AWS

In addition to deploying your services and tasks, you manage your config with `deploy` using the `config` subcommand.

Services

To see how your local values compare vs the current values of the service config in AWS, run:

```
deploy service config diff hello-world-test
```

To view your current values of the service config in AWS, run:

```
deploy service config show hello-world-test
```

To update the values of the service config to AWS, run:

```
deploy service config write hello-world-test
```

Tasks

To view your current values of the task config in AWS, run:

```
deploy task config show hello-world-test
```

To update the values of the task config to AWS, run:

```
deploy task config write hello-world-test
```

4.5.2 Reading From The Environment

In practice, you do not want the `deployfish.yml` file to contain actual passwords, so the best practice is to have the secret parameter values defined in an environment variable. You would then change the *config* section to be:

```
...
config:
  - VAR1=value1
  - VAR2=value2
  - PASSWORD1:secure=${env.PASSWORD1}
  - PASSWORD2:secure=${env.PASSWORD2}
```

See the *Interpolation* section for full details on how environment variable replacement in `deployfish.yml` works.

You typically should use a different file for each service.

4.5.3 Loading config: variables into your container environment

So now that we have all of these values loaded into the AWS Parameter Store, how do we use them? You need an execution role.

You must provide an execution_role that has permission to get the parameter store values, then your task or service will automatically have the parameter store values inserted into the environment.

4.6 Service Helper Tasks

In the `tasks` section of the service defintion, you can define helper tasks to be associated with your service and define commands on them that you can run via `deploy service task run <service> <command>`.

The reason this exists is to enable us to run one-off or periodic functions (migrate datbases, clear caches, update search indexes, do database backups or restores, etc.) for our services.

Task definitions listed in the `tasks` list support the same configuration options as those in the `services` list: `family`, `environment`, `network_mode`, `task_role_arn`, and all the same options under `containers`.

4.6.1 Example

When you do a `deploy service update <service_name>`, deployfish automaticaly updates the task definition to what is listed in the `tasks` entry for each task, and adds a docker label to the first container of the task definition for the service for each task, recording the `<family>:<revision>` string of the correct task revision.

```
services:
  - name: foobar-prod
    environment: prod
    cluster: foobar-prod-cluster
    count: 2
    service_role_arn: arn:aws:iam::123142123547:role/ecsServiceRole
    load_balancer:
      load_balancer_name: foobar-prod-elb
      container_name: foobar
      container_port: 80
    family: foobar-prod
```

(continues on next page)

(continued from previous page)

```
    network_mode: bridge
    task_role_arn: arn:aws:iam::123142123547:role/myTaskRole
    execution_role: arn:aws:iam::123142123547:role/myExecutionRole
    containers:
      - name: foobar
        image: foobar:0.0.1
        cpu: 128
        memory: 512
        ports:
          - "80"
          - "443"
        environment:
          - ENVIRONMENT=prod
          - SECRETS_BUCKET_NAME=my-secrets-bucket
    tasks:
      - launch_type: FARGATE
        network_mode: awsvpc
        vpc_configuration:
          subnets:
            - subnet-1234
            - subnet-1235
          security_groups:
            - sg-12345
        schedule_role: arn:aws:iam::123142123547:role/ecsEventsRole
        containers:
          - name: foobar
            cpu: 128
            memory: 256
        commands:
          - name: migrate
            containers:
              - name: foobar
                command: ./manage.py migrate
          - name: update_index
            schedule: cron(5 * * * ? *)
            containers:
              - name: foobar
                command: ./manage.py update_index
```

This example defines 2 separate new task defintions ("foobar-prod-tasks-migrate" and "foobar-prod-tasks-update-index") for our service "foobar-prod". Those two task definitions implement the two available commands on our service: `migrate` and `update_index`. These task defintions are created by starting with the Service's task definition, updating it with values from the top of the *tasks:* entry, and then further updating that with command specific setting for each of the commands in the `commands:` section.

When you do `deploy service update foobar-prod`, deployfish will create a new task defintion for each of the helper tasks and store their specific family:revision as tasks on the Service's task definition. Then when you run `deploy service task run foobar-prod migrate`, deployfish will:

1. Search for `migrate` among all the separate `commands` listings under `tasks` 1. Determine that `migrate` belongs to the `foobar-tasks-prod` task 1. Look on the active `foobar-prod` service task definition for the `edu.caltech.foobar-helper-prod` docker label 1. Use the value of that label to figure out which revision of our task to run. 1. Call the ECS `RunTasks` API call with that task revision.

4.7 Variable interpolation in deployfish.yml

You can use variable replacement in your service definitions to dynamically replace values from two sources: your local shell environment and from a remote terraform state file.

4.7.1 Environmnent variable replacement

You can add `${env.<environment var>}` to your service definition anywhere you want the value of the shell environment variable `<environment var>`. For example, for the following `deployfish.yml` snippet:

```
services:
  - name: foobar-prod
    environment: prod
    config:
      - MY_PASSWORD=${env.MY_PASSWORD}
```

`deployfish` does not by default inherit your shell environment when doing these `${env.VAR}` replacements. You must tell `deployfish` how you want it to load those environment variables.

deploy –import_env command line option

If you run `deploy` with the `--import_env` option, it will import your shell environment into the deployfish environment. Then anything you've defined in your shell environment will be available for `${env.VAR}` replacements.

Example:

```
deploy --import_env <subcommand> [options]
```

deploy –env_file command line option

`deploy` also supports declaring environment variables in a file instead of having to actually have them set in your environment. The file should follow these rules:

- Each line should be in `VAR=VAL` format.
- Lines beginning with # (i.e. comments) are ignored.
- Blank lines are ignored.
- There is no special handling of quotation marks.

Example:

```
deploy --env_file=<filename> <subcommand> [options]
```

Then anything you've defined in `<filename>` defined in your shell environment will be available for `${env.VAR}` replacements.

The "env_file" service definition option

You can also specify this environment variable file in the ECS service definition itself:

```
services:
  - name: hello-world-test
    cluster: hello-world-cluster
    count: 1
    family: hello-world
    env_file: config.env
    ...
```

4.7.2 Terraform variable replacment

If you're managing your AWS resources for your service with Terraform and you export your Terraform state files to S3, or if you are using Terraform Enterprise, you can use the values of your terraform outputs as string, list, or map values in your service definitions.

To do so, first declare a `terraform` top level section in your `deployfish.yml` file:

```
terraform:
  statefile: 's3://terraform-remote-state/my-service-terraform-state'
  lookups:
    ecs_service_role: 'ecs-service-role'
    cluster_name: '{service-name}-ecs-cluster-name'
    elb_name: '{service-name}-elb-name'
    storage_bucket: 's3-{environment}-bucket'
    task_role_arn: '{service-name}-task-role-arn'
    ecr_repo_url: 'ecr-repository-url'
```

If using Terraform Enterprise you need to provide the `workspace` and `organization` in place of the statefile:

```
terraform:
  workspace: sample_workspace
  organization: sampleOrganization
  lookups:
    ecs_service_role: 'ecs-service-role'
    cluster_name: '{service-name}-ecs-cluster-name'
    elb_name: '{service-name}-elb-name'
    storage_bucket: 's3-{environment}-bucket'
    task_role_arn: '{service-name}-task-role-arn'
    ecr_repo_url: 'ecr-repository-url'
    security_groups: '{service-name}-security-groups'
    subnets: 'service-subnets'
```

Then, wherever you have a string, list, or map value in your service definition, you can replace that with a terraform lookup, like so:

```
services:
  - name: my-service
    cluster: ${terraform.cluster_name}
    environment: prod
    count: 2
```

(continues on next page)

(continued from previous page)

```
    service_role_arn: ${terraform.ecs_service_role}
    load_balancer:
      load_balancer_name: ${terraform.elb_name}
      container_name: my-service
      container_port: 80
    family: my-service
    network_mode: bridge
    task_role_arn: ${terraform.task_role_arn}
    vpc_configuration:
      security_groups: ${terraform.security_groups}
      subnets: ${terraform.subnets}
    containers:
      - name: my-service
        image: ${terraform.ecr_repo_url}:0.1.0
        cpu: 128
        memory: 256
        ports:
          - "80"
        environment:
          - S3_BUCKET=${terraform.storage_bucket}
```

statefile

(String, Required) The `s3://` URL to your state file. For example, `s3//my-statefile-bucket/my-statefile`.

lookups

(Required) A dictionary of key value pairs where the keys will be used when doing string replacements in your service definition, and the values should evaluate to a valid terraform output in your terraform state file.

You can use these replacements in the values:

- `{environment}`: replace with the value of the `environment` option for the current service
- `{service-name}`: replace with the name of the current service
- `{cluster-name}`: replace with the name of the cluster for the current service

These values are evaluated in the context of each service separately.

profile

(String, Optional) The name of the AWS CLI Named Profile to use when retrieving the statefile from S3.

See Named Profiles

region

(String, Optional) The AWS region in which your S3 bucket lives.

workspace

(String, Required Terraform Enterprise) The Terraform Enterprise workspace.

organization

(String, Required Terraform Enterprise) The Terraform Enterprise organization.

–tfe_token option

In order to authenticate against terraform enterprise and read the state, you need to provide an API token. This can be either a user API token, team API token, or organization token.

```
deploy --tfe_token <token> <subcommand> [options]
```

It will also work if you specify an `ATLAS_TOKEN` environment variable while using the `--import_env` option.

```
deploy --import_env <subcommand> [options]
```

4.8 Advanced Usage: using a different AWS Profile for the statefile

It is not uncommon to of your Terraform state files in a single bucket, even if the associated Terraform templates affect resources in many different accounts.

If this is the case with you, you can specify which AWS Credentials named profile (see Named Profiles for more information). use to retrieve the state files by adding the `profile` and `region` settings to your `terrraform:` section:

```
terraform:
  statefile: 's3://hello-world-remotestate-file/hello-world-terraform-state'
  profile: configs
  region: us-west-2
  lookups:
    cluster_name: '{environment}-cluster-name'
    load_balancer_name: '{environment}-elb-id'
    task_role_arn: 'iam-role-hello-world-{environment}-task'
    rds_address: '{environment}-rds-address'
    app_bucket: 's3-hello-world-{environment}-bucket'
```

This will tell `deployfish` that, for retrieving this statefile only, it should use the "configs" AWS profile.

CHAPTER

FIVE

COMMAND-LINE REFERENCE

CHAPTER

SIX

EXTENDING DEPLOYFISH

6.1 Introduction

deployfish has a modular architecture that allows you to add subcommands that have access to the internal objects through the *deployfish* library. As an example, you can look at deployfish-mysql.

To get started, you'll need to create a new Click command group:

```
import click
import os

from deployfish.cli import cli
from deployfish.core.models import Service
from deployfish.config import Config, needs_config

@cli.group(short_help="Manage a remote MySQL database")
def mysql():
    pass
```

You can then add commands to that group:

```
@mysql.command('create', short_help="Create database and user")
@click.pass_context
@click.argument('identifier')
@needs_config
def create(ctx, identifier):
    service = Service.objects.get(identifier)

    host, name, user, passwd, port = _get_db_parameters(service)
    root = click.prompt('DB root user')
    rootpw = click.prompt('DB root password')

    cmd = "/usr/bin/mysql --host={} --user={} --password={} --port={} --execute=\"create␣
↪database {}; grant all privileges on {}.* to '{}'@'%' identified by '{}';\"".
↪format(host, root, rootpw, port, name, name, user, passwd)

    success, output = service.run_remote_script([cmd])
    print success, output
```

As you can see, you have full access to the *Service* class.

To register your commands with *deployfish*, you'll add an *entry_points* entry in your *setup.py* file:

```
entry_points={
    'deployfish.command.plugins': [
        'mysql = deployfish_mysql.mysql'
    ]
},
```

Then install your library with *pip*.

6.2 Loading a Model from configuration in deployfish.yml

Classes derived from *deployfish.core.models.abstract.Model* can be configured from configuration in deployfish.yml.

1. Extract the configuration stanza for your object from deployfish.yml:

   item_config = Config.get_section_item('my_section_name', 'my_item_name')

2. Generate your configued Model subclass instance by doing:

   instance = MyModel.new(item_config, 'deployfish')

`MyModel.new()` does this:

1. Find the proper `deployfish.core.adapter.abstract.Adapter` subclass that will translate between `item_config` and properly configured `data` for `MyModel` by looking in the adapter registry `deployfish.registry.importer_registry`. This registry maps `Adapter` subclasses to `deployfish.core.models.abstract.Model` subclasses.
2. Instantiate the `Adapter` subclass, passing in our `item_config` to its constructor.
3. Run `MyAdapter.convert()`. This will generate `data`, a dict formatted to look like what boto3's `describe_*` API method would return for the `MyModel`, and `kwargs`, extra configuration `MyModel` may need in order to function properly.
4. Instantiate a `MyModel` by doing:

   instance = MyModel.__init__(data)

5. Set any other necessary attributes on `instance` from the data we returned above in `kwargs`.

Note: One of the challenges we have in constructing `MyModel` from deployfish.yml is that we need to ensure we can also load `MyModel` purely from AWS calls. When loading an object from AWS , we want any dependent objects (e.g. a Service's TaskDefinition) to be lazy loaded from AWS in order to reduce the API calls to only the data we need at the moment –, this saves the user from having to wait too long. When loading an object from deployfish.yml however, we load all the dependent objects at the same time have to provide them to the Model instance all at once, with no lazy loading.

Largely we do this with @property and @propety.setter properties. The main @property laods the data from AWS if necessary, while the @property.setter circumvents the AWS loading.

---

1. Create a subclass of *deployfish.core.adapters.abstract.Adapter* * The `.__init__()` for your subclass will get passed the deployfish.yml configuration for your object, and will store it as *self.data* * Override *.convert()* on that subclass to use *self.data* to generate *data*, a dict that replicates what boto3 would return were we to call the *describe_** method for that object, and *kwargs*, keyword arguments for the object's *.new()* factory method (described below)

...

6.3 Loading a Service from deployfish.yml

First create all the appropriate objects from the service config in deployfish.yml.

`deployfish.core.adapters.deployfish.ServiceAdapter`, does this, in this order:

1. Build the data necessary for the `data` parameter to `Service.__init__()` from the service's config.
2. If a `config:` section is present in the service's config, load the list of `Secrets` from the service's `config:` section via `deployfish.core.adapters.deployfish.SecretAdapter` and `deployfish.core.adapters.deployfish.ExternalSecretAdapter`.
3. Use `deployfish.core.adapters.deployfish.TaskDefinitionAdapter` to create a `TaskDefinition` from the service config. This needs the `Secrets` we created above, if any.
4. If `application_scaling:` section is present in the service's config, build the ApplicationScaling objects, which are: * `ScalableTarget` (from `deployfish.core.adapters.deployfish.ECServiceScalableTargetAdapter`) * One or more `ScalingPolicy` objects (via `deployfish.core.adapters.deployfish.ECServiceScalingPolicyAdapter`) * One `CloudwatchAlarm` per ScalingPolicy (via `deployfish.core.adapters.deployfish.ECServiceCPUAlarmAdapter`)

- If a `service_discovery:` section is present in the service's config, build a `ServiceDiscoveryService` object (via `deployfish.core.adapters.deployfish.ServiceDiscoveryServiceAdapter`).
- If a `tasks:` section is present in the service's config, build configuration for one or more `ServiceHelperTasks` (via `deployfish.core.adapters.deployfish.ServiceHelperTaskAdapter`, but (**important**) loaded in Service.new(), not in `ServiceAdapter.convert()` – we need the fully configured `Service` object in order to make the helper tasks, and that doesn't happen until we get into `Service.new()`.

Finally the `Service` object is configured.

Here's how `Service.save()` works when creating a service:

- If we have `ServiceHelperTasks`, create them in AWS and save their `family:revisions` on our `TaskDefinition`, so that we know which specific revision to run to get the version of the code we want
- Create the `TaskDefinition` in AWS, and save its ARN to the Service as `taskDefinition`
- If we need it, create the `ServiceDiscoveryService` in AWS, and save its ARN to the service as `serviceRegistries[0]['registryArn']`; otherwise delete any `ServiceDiscoveryService` associated with the `Service`.
- Create the `Service` in AWS
- If we need it, create the `ScalingTarget`, `ScalingPolicy` and `CloudwatchAlarm` objects in AWS, otherwise delete any such that exist in AWS

6.4 Elastic Container Service

6.4.1 Service

6.4.2 Tasks

6.4.3 TaskDefinition

6.4.4 Cluster

6.5 Application/Network Load Balancing

6.6 AWS SSM Paramter Store