

Database & Container Deployment

Software Architecture

March 25, 2025

Evan Hughes & Brae Webb



Aside

Github Classroom links for this practical can be found on Edstem <https://edstem.org/au/courses/15375/discussion/1753712>

1 This Week

This week we are going to deploy our todo application, now called TaskOverflow, on AWS infrastructure using a hosted database and a single server website.

Specifically, this week you need to:

- Deploy an AWS Relational Database Service (RDS) using Terraform.
- Deploy the TaskOverflow container on AWS infrastructure using an ECS cluster.

2 Terraform in AWS Learner Labs

Following the steps from the week four practical, start a Learner Lab in AWS Academy. For this practical, you do not need to create any resources using the AWS Console. The console can be used to verify that Terraform has correctly provisioned resources.

1. Using the GitHub Classroom link for this practical provided by your tutor on edstem, create a repository to work within.
2. Clone the repository or open an environment in GitHub CodeSpaces¹
3. Start the Learner Lab then, once the lab has started, click on ‘AWS Details’ to display information about the lab.



4. Click on the first ‘Show’ button next to ‘AWS CLI’ which will display a text block starting with [default].
5. Within your repository create a `credentials` file and copy the contents of the text block into the file. **Do not share this file contents — do not commit it.**
6. Create a `main.tf` file in the your repository with the following contents:

```
> cat main.tf

terraform {
    required_providers {
        aws = {
            source = "hashicorp/aws"
            version = "~> 5.0"
        }
    }

    provider "aws" {
        region = "us-east-1"
        shared_credentials_files = ["./credentials"]
    }
}
```

¹If you are using CodeSpaces, you will need to reinstall Terraform using the same steps as last week.

7. We need to initialise terraform which will fetch the required dependencies. This is done with the `terraform init` command.

```
$ terraform init
```

This command will create a `.terraform` directory which stores providers and a provider lock file, `.terraform.lock.hcl`.

8. To verify that we have setup Terraform correctly, use `terraform plan`.

```
$ terraform plan
```

As we currently have no resources configured, it should find that no changes are required. Note that this does not ensure our credentials are correctly configured as Terraform has no reason to try authenticating yet.

3 Deploying a Database in AWS

Warning

This section manually deploys a PostgreSQL RDS instance, this is intended as a demonstration by your tutor. You should attempt to deploy your infrastructure using Terraform rather than manually.

To get started let us jump into the lab environment and have a look at AWS RDS which is an AWS managed database service. To get to the RDS service either search it or browse Services -> Database -> RDS as shown below.



Now we are in the management interface for all our RDS instances. Head to “DB Instances (0/40)” or click “Databases” on the left panel.

The screenshot shows the Amazon RDS Management Console. The top navigation bar includes the AWS logo, services dropdown, search bar, and account information (N. Virginia, vocabs/user1718566=Test_Student @ 7287-7499-1353). The left sidebar has a header "Amazon RDS" and sections: Dashboard, Databases, Query Editor, Performance insights, Snapshots, Automated backups, Reserved instances, Proxies, Subnet groups, Parameter groups, Option groups, Custom Availability Zones, Custom engine versions, Events, and Event subscriptions. The main content area is titled "Resources" and displays usage statistics for the US East (N. Virginia) region: DB Instances (0/40), Parameter groups (0), Option groups (0), DB Clusters (0/40), Reserved instances (0/40), Snapshots (0), and Event subscriptions (0/20). A "Create database" section is also present. On the right, there's a "Recommended for you" sidebar with links to Time-Series Tables in PostgreSQL, Implementing Cross-Region DR, Amazon RDS Backup and Restore using AWS Backup, and Build RDS Operational Tasks.

This page should appear familiar as it is very similar to the AWS EC2 instance page. Let us create a new database by hitting the “Create Database” button.

The screenshot shows the "Databases" page under the RDS service. The top navigation bar shows "RDS > Databases". The main header is "Databases" with options: Group resources (selected), Modify, Actions, Restore from S3, and Create database. Below is a search bar with placeholder "Filter by databases". The table header includes columns: DB identifier, Role, Engine, Region & AZ, Size, Status, and CPU. A message at the bottom states "No instances found".

Warning

In the next section we cannot use the Easy Create option as it tries to create a IAM account which is disabled in Learner Labs.

We will be creating a standard database so select standard and PostgreSQL. We will use version 14, which is a fairly recent release.

Choose a database creation method [Info](#)

Standard create

You set all of the configuration options, including ones for availability, security, backups, and maintenance.

Easy create

Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

Engine options

Engine type [Info](#)

Aurora (MySQL Compatible)



Aurora (PostgreSQL Compatible)



MySQL



MariaDB



PostgreSQL



Oracle

ORACLE®

Microsoft SQL Server



Engine Version

PostgreSQL 14.6-R1



For today we are going to use “Free Tier” but in the future, you may wish to explore the different deployment options. Please peruse the available different options.

Templates

Choose a sample template to meet your use case.

Production

Use defaults for high availability and fast, consistent performance.

Dev/Test

This instance is intended for development use outside of a production environment.

Free tier

Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS.

[Info](#)

Availability and durability

Deployment options [Info](#)

The deployment options below are limited to those supported by the engine you selected above.

Single DB instance (not supported for Multi-AZ DB cluster snapshot)

Creates a single DB instance with no standby DB instances.

Multi-AZ DB instance (not supported for Multi-AZ DB cluster snapshot)

Creates a primary DB Instance and a standby DB Instance in a different AZ. Provides high availability and data redundancy, but the standby DB Instance doesn't support connections for read workloads.

Multi-AZ DB Cluster - new

Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.

Now we need to name our database and create credentials to use when connecting from our application. Enter memorable credentials as these will be used later.

Settings

DB instance identifier [Info](#)

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

todo

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username [Info](#)

Type a login ID for the master user of your DB instance.

todo

1 to 16 alphanumeric characters. First character must be a letter.

Manage master credentials in AWS Secrets Manager

Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

ⓘ If you manage the master user credentials in Secrets Manager, some RDS features aren't supported.
[Learn more](#) ↗

Auto generate a password

Amazon RDS can generate a password for you, or you can specify your own password.

Master password [Info](#)

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm master password [Info](#)

For exploring the process select t2.micro, which should be adequate for our needs.

Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class [Info](#)

- Standard classes (includes m classes)
- Memory optimized classes (includes r and x classes)
- Burstable classes (includes t classes)

db.t4g.micro
2 vCPUs 1 GiB RAM Network: 2,085 Mbps

Include previous generation classes

For storage we will leave all the default options.

Storage

Storage type [Info](#)

General Purpose SSD (gp2)
Baseline performance determined by volume size

Allocated storage

20

GiB

(Minimum: 20 GiB. Maximum: 16,384 GiB) Higher allocated storage [may improve](#) IOPS performance.

 You might see better baseline performance with your selected volume size by specifying General Purpose SSD storage. [Learn more about using Provisioned IOPS storage for consistent performance.](#)

Storage autoscaling [Info](#)

Provides dynamic scaling support for your database's storage based on your application's needs.

Enable storage autoscaling

Enabling this feature will allow the storage to increase once the specified threshold is exceeded.

Maximum storage threshold [Info](#)

Charges will apply when your database autoscales to the specified threshold

1000

GiB

Minimum: 21 GiB. Maximum: 16,384 GiB

In connectivity we need to make sure our instance is publicly available. Usually you do not want to

expose your databases publicly and, would instead, have a web server sitting in-front. For our learning purposes though we are going to expose it directly just like we did with our EC2 instances early in the course.

When selecting public access as yes we have to create a new Security Group, give this Security Group a sensible name.

DB subnet group [Info](#)

Choose the DB subnet group. The DB subnet group defines which subnets and IP ranges the DB instance can use in the VPC that you selected.

default ▾

Public access [Info](#)

Yes

RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.

No

RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) [Info](#)

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

Choose existing

Choose existing VPC security groups

Create new

Create new VPC security group

Existing VPC security groups

Choose one or more options ▾

default X

Availability Zone [Info](#)

No preference ▾

RDS Proxy

RDS Proxy is a fully managed, highly available database proxy that improves application scalability, resiliency, and security.

Create an RDS Proxy [Info](#)

RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. RDS Proxy has additional costs. For more information, see [Amazon RDS Proxy pricing](#).

Certificate authority - optional [Info](#)

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

rds-ca-2019 (default) ▾

If you don't select a certificate authority, RDS chooses one for you.

▼ Additional configuration

Database port [Info](#)

TCP/IP port that the database will use for application connections.

5432 ▾

We will leave the authentication as password based but we need to expand the “Additional configuration”. Fill in the “Initial Database Name” section to be “todo”, this will automatically create the database that our todo application expects to connect to.

▼ **Additional configuration**
Database options, encryption turned off, backup turned off, backtrack turned off, maintenance, CloudWatch Logs, delete protection turned off.

Database options

Initial database name [Info](#)

If you do not specify a database name, Amazon RDS does not create a database.

DB parameter group [Info](#)

Option group [Info](#)

Backup

Enable automated backups
Creates a point-in-time snapshot of your database

Encryption

Enable encryption
Choose to encrypt the given instance. Master key IDs and aliases appear in the list after they have been created using the AWS Key Management Service console. [Info](#)

Log exports
Select the log types to publish to Amazon CloudWatch Logs

PostgreSQL log
 Upgrade log

IAM role
The following service-linked role is used for publishing logs to CloudWatch Logs.

Now we can click create which will take some time.

Estimated monthly costs

The Amazon RDS Free Tier is available to you for 12 months. Each calendar month, the free tier will allow you to use the Amazon RDS resources listed below for free:

- 750 hrs of Amazon RDS in a Single-AZ db.t2.micro Instance.
- 20 GB of General Purpose Storage (SSD).
- 20 GB for automated backup storage and any user-initiated DB Snapshots.

[Learn more about AWS Free Tier.](#)

When your free usage expires or if your application use exceeds the free usage tiers, you simply pay standard, pay-as-you-go service rates as described in the [Amazon RDS Pricing page](#).

 You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.

Cancel

Create database

It may take 10 to 30 minutes to create. The database will also do a initial backup when its created.

Databases		<input checked="" type="checkbox"/> Group resources		Modify	Actions ▾	Restore from S3	Create database	
		<input type="text"/> todo X < 1 > ⚙️						
	DB identifier	▲	Role ▾	Engine ▾	Region & AZ ▾	Size ▾	Status ▾	Actions ▾
	todo		Instance	PostgreSQL	-	db.t4g.micro	 Creating	-

When the database has finished being created you can select it to view the configuration and details. In this menu we also see the endpoint address which we will need to configure our TaskOverflow application to use.

todo

[Modify](#)[Actions ▾](#)

Summary

DB identifier
todo

CPU

-

Status

Available

Class

db.t4g.micro

Role
Instance

Current activity

 0 Connections

Engine

PostgreSQL

Region & AZ

us-east-1d

[Connectivity & security](#)[Monitoring](#)[Logs & events](#)[Configuration](#)[Maintenance & backups](#)[Tags](#)

Connectivity & security

Endpoint & port

Endpoint
todo.csf64hdqthj.us-
east-1.rds.amazonaws.com

Networking

Availability Zone
us-east-1d

Security

VPC security groups
default (sg-08d05b608beff2f02)
Active

4 RDS Database with Terraform

Now would be a good time to browse the documentation for the RDS database in Terraform. You will want to get practice at reading and understanding Terraform documentation.

https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/db_instance

Using our manual configuration, we can come up with a resource with the appropriate parameters as below:

```
``` hcl
locals {
 database_username = "administrator"
 database_password = "foobarbaz" # this is bad
}

resource "aws_db_instance" "taskoverflow_database" {
 allocated_storage = 20
 max_allocated_storage = 1000
 engine = "postgres"
 engine_version = "14"
 instance_class = "db.t4g.micro"
 db_name = "todo"
 username = local.database_username
 password = local.database_password
 parameter_group_name = "default.postgres14"
 skip_final_snapshot = true
 vpc_security_group_ids = [aws_security_group.taskoverflow_database.id]
 publicly_accessible = true

 tags = {
 Name = "TaskOverflow"
 }
}
```

```

```

    Name = "taskoverflow_database"
}
}

```

When we created the database using the AWS Console, we needed an appropriate security group so that we could access the database. We can create the security group using Terraform as well.

```

> cat main.tf

resource "aws_security_group" "taskoverflow_database" {
  name = "taskoverflow_database"
  description = "Allow inbound Postgresql traffic"

  ingress {
    from_port = 5432
    to_port = 5432
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = [":/:0"]
  }

  tags = {
    Name = "taskoverflow_database"
  }
}

```

5 Container on AWS

As we mentioned in the Infrastructure as Code notes [1], in this course we will use Docker to configure machines and Terraform to configure infrastructure. AWS has the ability to deploy Docker containers using a service known as Elastic Container Service (ECS). We will cover ECS and deploying manually via EC2 so you can use the method you feel most comfortable with.

For this practical we have made available a Docker container running the TaskOverflow application which you can use for your AWS deployment. This container is available on GitHub under the CSSE6400 organisation:

<https://ghcr.io/csse6400/taskoverflow:latest>

This container is very similar to what you have been building in the practicals but contains a simple UI and some extra features for the future practicals.²

²If you are interested, the source code is available on GitHub <https://github.com/csse6400/practical>

5.1 Setup

Of all the different ways that we can deploy our application, we have decided to offload the database to AWS RDS. This means that we can move all the "state" of our application away from our containerised environment.

To begin, we will reuse our Terraform from above for deploying the RDS database. Extend the existing local Terraform variables to include the address of the container, such that we have:

```
» cat main.tf

locals {
  image = "ghcr.io/csse6400/taskoverflow:latest"
  database_username = "administrator"
  database_password = "foobarbaz" # this is bad
}
```

This already sets up an RDS instance of Postgres and a security group to allow access to it. Now we can run `terraform init` and `terraform apply` to create our database.

We have also added a local variable for us to use later. Variables in Terraform can be populated via two mechanisms, they can be in a `variables` block which can be overridden, or they can be in a `locals` block which can be used to store values that are used in multiple places.

Now we will use ECS to deploy a containerised version of our application.

5.2 [Path A] EC2

Aside

In 2024 we have removed the EC2 pathway from this practical but we have left the deployment diagram to allow comparisons to ECS. Please skip to Section 5.3 for the ECS approach.

TaskOverflow on EC2 (Deployment Diagram)



Legend
container

5.3 [Path B] ECS

Aside

This is the recommended path for the course and is the path that we will be suggesting for the future.



Legend
container

Congratulations! You have chosen to go down the ECS path which mimics a similar environment as Docker Compose but as an AWS service. This path is new for the course this year so please let your tutors know of any issues you have.

To start off we need to get some information from our current AWS environment so that we can use it later. Add the below to fetch the IAM role known as LabRole which is a super user in the Learner Lab environments which can do everything you can do through the UI. We will also be fetching the default VPC and the private subnets within that VPC as they are required for the ECS network configuration.

```
data "aws_iam_role" "lab" {
  name = "LabRole"
}

data "aws_vpc" "default" {
  default = true
}

data "aws_subnets" "private" {
```

```

filter {
  name = "vpc-id"
  values = [data.aws_vpc.default.id]
}
}

```

In Terraform, the way to retrieve external information is data sources. These are functionally like resources but they are not created or destroyed, instead they are populated with attributes from the current state. See the below for the minor syntactic difference.

```

data "aws_iam_role" "lab" {
  ...
}

resource "aws_db_instance" "database" {
  ...
}

```

Now that we have access to the information required, we can create the ECS cluster to host our application.

The first step is to create the ECS cluster which is just a logical grouping of any images. All that is required is a name for the new grouping.

```

» cat main.tf
resource "aws_ecs_cluster" "taskoverflow" {
  name = "taskoverflow"
}

```

On its own this cluster is not particularly useful. We need to create a task definition which is a description of the container that we want to run. This is where we will define the image that we want to run, the environment variables, the port mappings, etc. This is similar to a server entry in Docker Compose.

Warning

The «DEFINITION line cannot have a trailing space. Ensure that one has not been erroneously inserted.

```

» cat main.tf
resource "aws_ecs_task_definition" "taskoverflow" {
  family = "taskoverflow"
  network_mode = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu = 1024
  memory = 2048
  execution_role_arn = data.aws_iam_role.lab.arn
}

```

```

    container_definitions = <<DEFINITION
[
{
  "image": "${local.image}",
  "cpu": 1024,
  "memory": 2048,
  "name": "todo",
  "networkMode": "awsvpc",
  "portMappings": [
    {
      "containerPort": 6400,
      "hostPort": 6400
    }
  ],
  "environment": [
    {
      "name": "SQLALCHEMY_DATABASE_URI",
      "value": "postgresql://${local.database_username}:${local.database_password}
        @${aws_db_instance.taskoverflow_database.address}:${aws_db_instance.
        taskoverflow_database.port}/${aws_db_instance.taskoverflow_database.
        db_name}"
    }
  ],
  "logConfiguration": {
    "logDriver": "awslogs",
    "options": {
      "awslogs-group": "/taskoverflow/todo",
      "awslogs-region": "us-east-1",
      "awslogs-stream-prefix": "ecs",
      "awslogs-create-group": "true"
    }
  }
}
]
DEFINITION
}

```

family A family is similar to the name of the task but it is a name that persists through multiple revisions of the task.

network_mode This is the network mode that the container will run in, we want to run on regular AWS VPC infrastructure.

requires_compatibilities This is the type of container that we want to run. This can be fargate, EC2, or external.

cpu The amount of CPU units that the container will be allocated. 1024 is equivalent to one vCPU.

memory The amount of memory that the container will be allocated, here we've chosen 2GB.

execution_role_arn The IAM role that the container will run as. Importantly, we have re-used the lab role we previously retrieved. This gives the instance full admin permission for our lab environment.

container_definitions This is the definition of the container, it should look very familiar to Docker Compose. The only additional feature here is the `logConfiguration`. This configures our container to write logs to AWS CloudWatch so that we can see if anything has gone wrong.

Now we have a description of our container as a task. We need a service to run the container on. This is functionally similar to an auto-scaling group from the lecture. We specify how many instances of the described container we want and it will provision them. We also specify which ECS cluster and AWS subnets to run the containers within.

```
> cat main.tf

resource "aws_ecs_service" "taskoverflow" {
  name = "taskoverflow"
  cluster = aws_ecs_cluster.taskoverflow.id
  task_definition = aws_ecs_task_definition.taskoverflow.arn
  desired_count = 1
  launch_type = "FARGATE"

  network_configuration {
    subnets = data.aws_subnets.private.ids
    security_groups = [aws_security_group.taskoverflow.id]
    assign_public_ip = true
  }
}
```

In the above we refer to a non-existent security group. As always, to be able to access our instances over the network we need to add a security group policy to enable it.

```
> cat main.tf

resource "aws_security_group" "taskoverflow" {
  name = "taskoverflow"
  description = "TaskOverflow Security Group"

  ingress {
    from_port = 6400
    to_port = 6400
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```

    egress {
      from_port = 0
      to_port = 0
      protocol = "-1"
      cidr_blocks = ["0.0.0.0/0"]
    }
}

```

Finally, if we run the appropriate `terraform init` and `terraform apply` commands, it should provision an ECS cluster with a service that will then create one ECS container based on our task description.

Note that we are doing something a bit weird in this deployment. Normally ECS expects multiple instances of containers, so it naturally expects a load balancer. This makes it difficult for us to discover the public IP of our single instance using Terraform. Instead, you will need to use the AWS Console to find the public IP address.

This is an opportunity for you to explore the ECS interface and find the task, within the service, within the cluster that we have provisioned.

5.4 [Path C] EKS / K8S

This path is not described in the course yet, but we recommend that if you liked the course to have a look at [Kubernetes³](#) as it is widely used in industry.

6 Hosting TaskOverflow Images

When we last deployed a container on AWS, we used an existing hosted image. Now, we will be developing our own image, so we will need a mechanism to host the image. For this, we will be using an AWS ECR, Docker, and Terraform. AWS ECR is the Elastic Container Registry, it is a container registry like DockerHub or GitHub. We can use it to host our images, using the process below:

1. Use Terraform to create an ECR repository for our image.
2. Use Terraform to build our Docker image.
3. Use Terraform to push our Docker image.

Info

This is a non-standard process. As you may have seen in the DevOps tutorial, we would ordinarily like our code commits to trigger a CI/CD pipeline which builds the images.

If you would like, you can use GitHub actions to build and push your container to the GitHub container registry and authenticate when you pull the image. However, using ECR simplifies the process, despite the oddities introduced by having a non-persistent ECR repository.

³<https://kubernetes.io/>

Getting Started

1. Using the GitHub Classroom link for this practical provided by your tutor on edstem, create a repository to work within.
2. Install Terraform if not already installed, as it will be required again this week.
3. Start your learner lab and copy the AWS Learner Lab credentials into a credentials file in the root of the repository.

What's New We are starting again with our todo application from roughly where we left off in the week 3 practical. We've added a new directory `todo/app` that has the static HTML files for the TaskOverflow website and added a route to serve these files. We have also created a production version of the server that uses gunicorn, the `bin` directory is used by this image. Our original Docker image is now in `Dockerfile.dev`.

We will setup our initial Terraform configuration. Note that now we introduce a new required provider. This provider is for Docker.

```
» cat main.tf

terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.0"
    }
    docker = {
      source = "kreuzwerker/docker"
      version = "3.0.2"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_files = ["./credentials"]
}
```

As with our AWS provider, when we initially configure the provider, we want to authenticate so that we can later push to our registry using the Docker provider. We will use the `aws_ecr_authorization_token` data block to get appropriate ECR credentials for Docker.

```
» cat main.tf

data "aws_ecr_authorization_token" "ecr_token" {}

provider "docker" {
  registry_auth {
    address = data.aws_ecr_authorization_token.ecr_token.proxy_endpoint
    username = data.aws_ecr_authorization_token.ecr_token.user_name
  }
}
```

```
    password = data.aws_ecr_authorization_token.ecr_token.password
}
}
```

We need to use Terraform to create an ECR repository to push to.

```
» cat main.tf

resource "aws_ecr_repository" "taskoverflow" {
  name = "taskoverflow"
}
```

The URL for containers in the ECR following the format below:

{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME}

Remember — to push to a container registry we need a local container whose tag matches the remote URL. We could then create and push the container locally with:

```
docker build -t {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME} .
docker push {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME}
```

However, it would be easier if we could build and push this container from within Terraform. We can use the Docker provider for this.

```
» cat image.tf

resource "docker_image" "taskoverflow" {
  name = "${aws_ecr_repository.taskoverflow.repository_url}:latest"
  build {
    context = "."
  }
}

resource "docker_registry_image" "taskoverflow" {
  name = docker_image.taskoverflow.name
}
```

Notice that we are able to utilize the output of the ECR repository as the URL which resolves to the correct URL for the image.

References

- [1] B. Webb, “Infrastructure as code,” March 2022. <https://csse6400.uqcloud.net/handouts/iac.pdf>.