Deploying with Terraform

Software Architecture

March 6, 2023

Evan Hughes & Brae Webb

1 Before Class

Ensure you've had practice using the AWS Academy learner lab. It's preferable if you already have terraform installed[1]. Please also have one of IntelliJ IDEA, PyCharm, or VSCode with the terraform plugin installed.

2 This Week

This week we are going to deploy TaskOverflow our Todo Application to AWS using a hosted database and a single server website.

Specifically, this week you need to:

- Authenticate Terraform to use the AWS learner lab.
- Configure an RDS database.

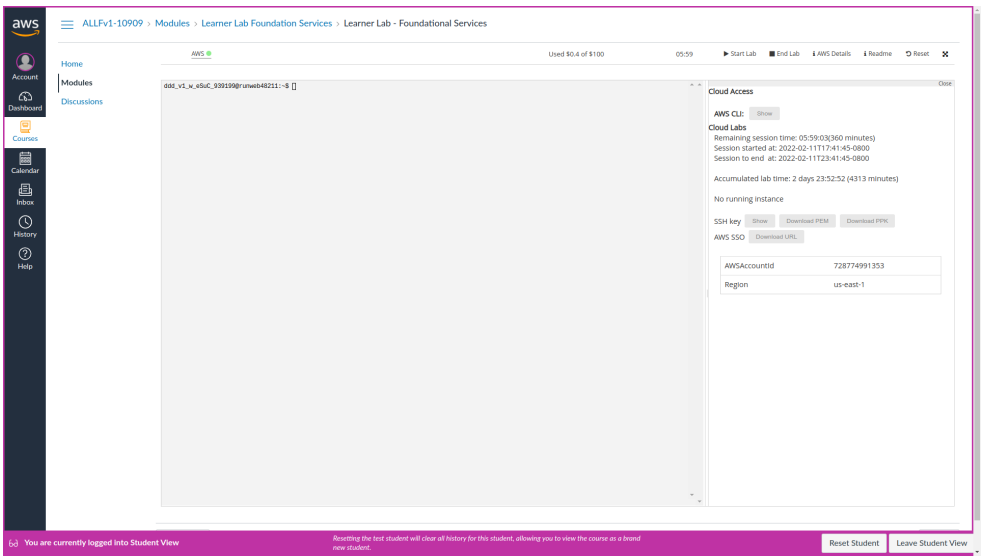
Path A Configure a single server and deploy the container.

Path B Configure a ECS cluster and deploy the container.

3 Using 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 in the AWS Console. The console can be used to verify that Terraform has correctly provisioned resources.

1. Once the learner lab has started, click on 'AWS Details' to display information about the lab.



[1] https://learn.hashicorp.com/tutorials/terraform/install-cli

2. Click on the first 'Show' button next to 'AWS CLI' which will display a text block starting with `[default]`.
3. Create a directory for this week's practical.
4. Within that directory 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.**
5. Create a `main.tf` file in the same directory with the following contents:

```
» cat main.tf
terraform {
    required_providers {
        aws = {
            source = "hashicorp/aws"
            version = "~> 4.0"
        }
    }
}

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

The `terraform` block specifies the required external dependencies, here we need to use the AWS provider. The `provider` block configures the AWS provider, instructing it which region to use and how to authenticate (using the credentials file we created).

6. 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`.

7. 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.

4 Deploying a Database in AWS

Info

This section manually deploys a Postgresql RDS instance, which is not the courses end goals but is a good way to get started with AWS. Latter this practical we will use terraform to create the database, so this section is optional and is better to be observed rather than actioned.

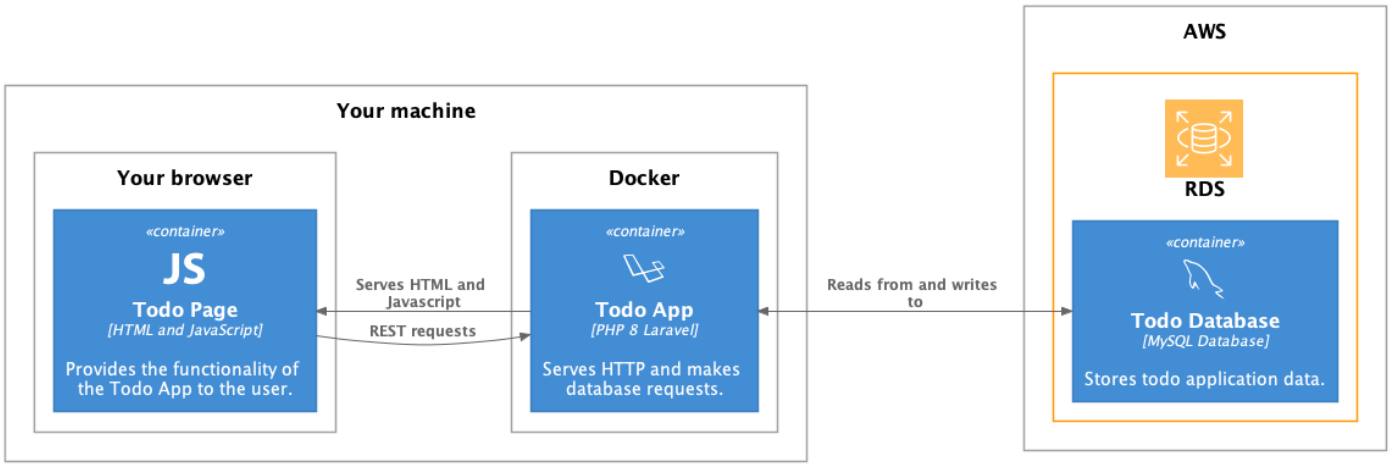
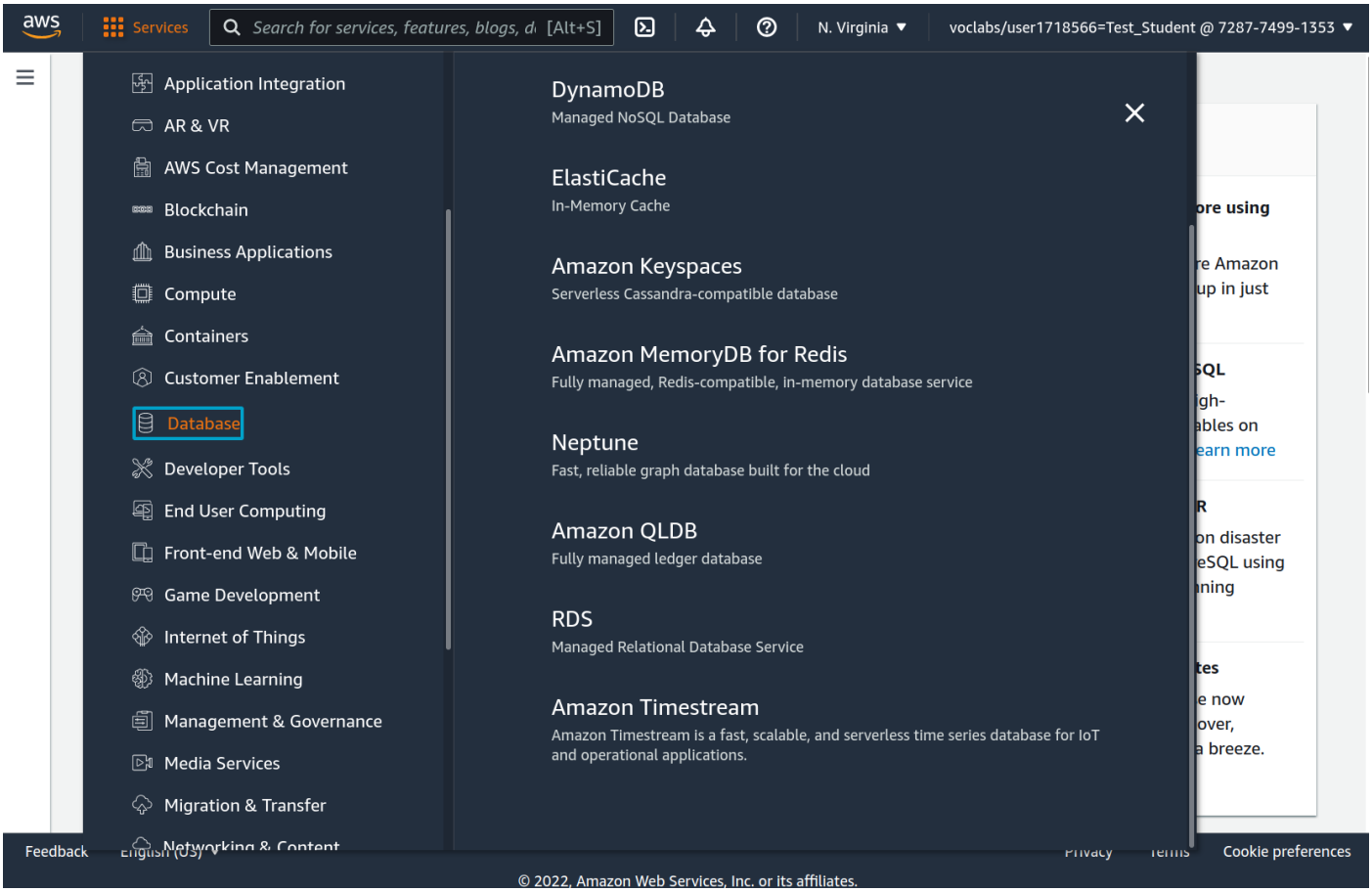


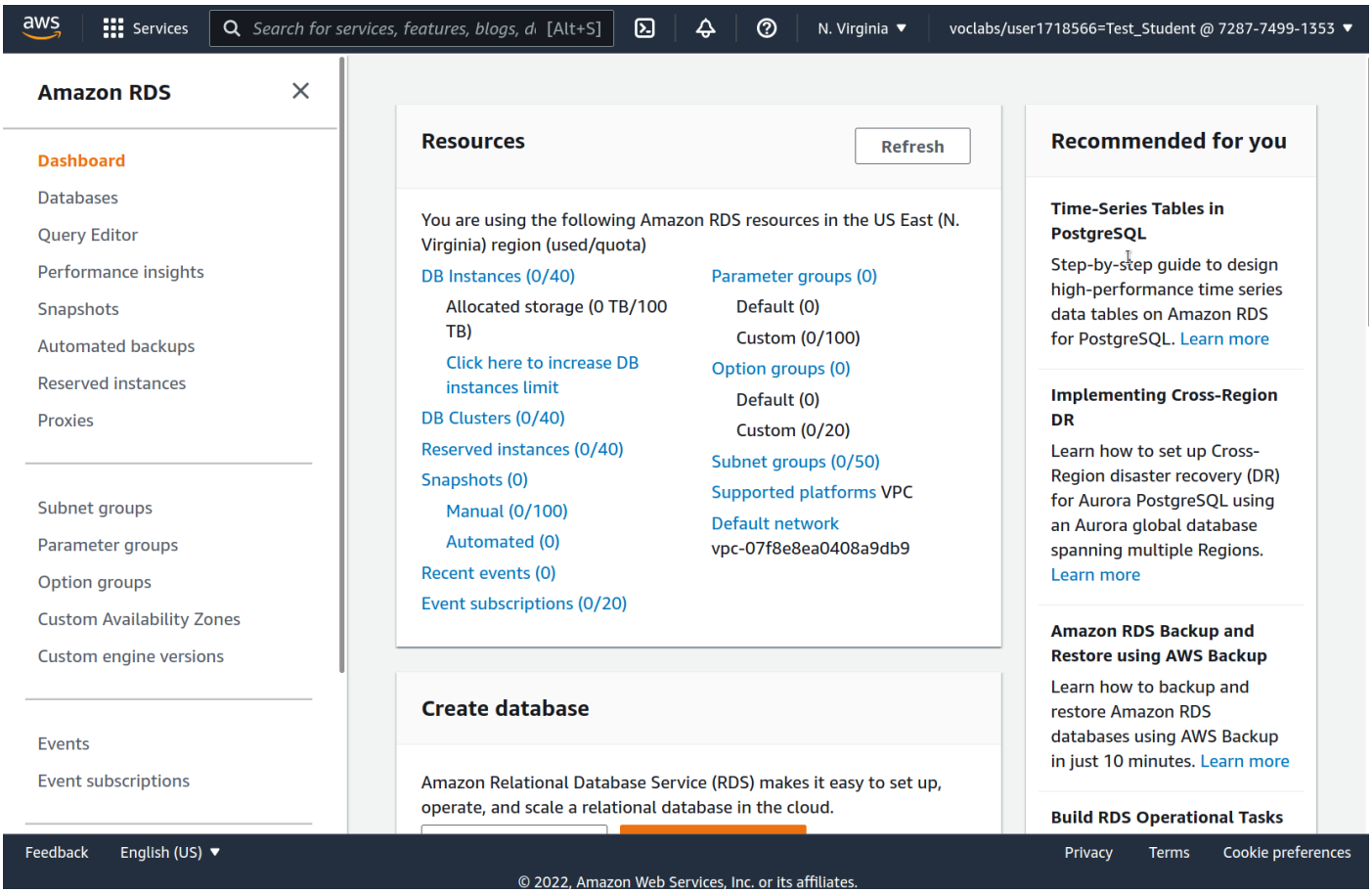
Figure 1: Remote database deployment diagram

This is the last time we will heavily use the AWS user interface in the practicals. If you already feel confident in the AWS environment skip this section and move on to the next one.

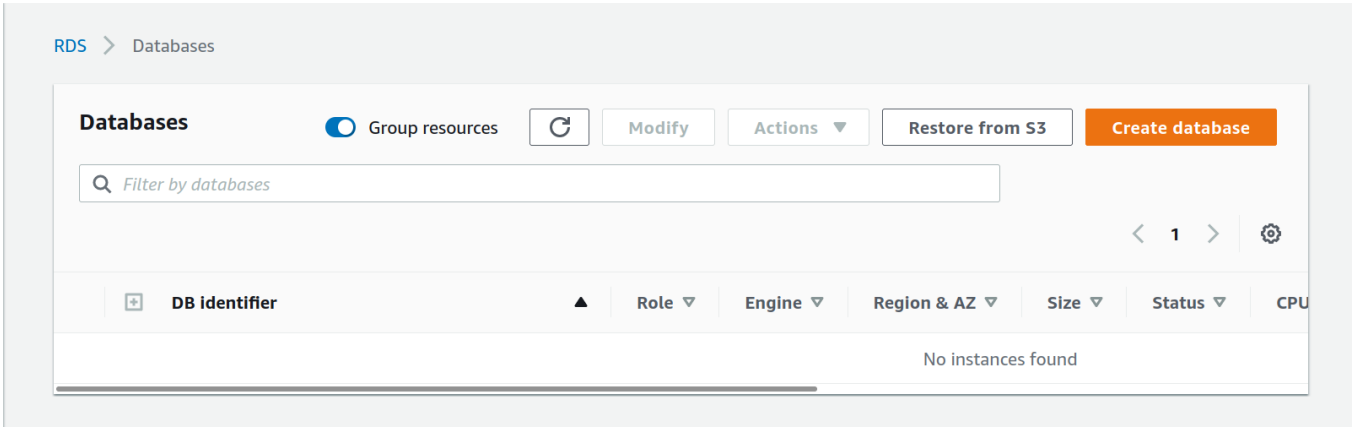
To get started let's jump into the lab environment and have a look at AWS RDS which is an AWS managed database service. To get to the RDS service either search it or browse Services -> Database -> RDS as shown below.



Now we are in the management page for all our database instances, for today we just want to get a small instance running to explore the service. Head to “DB Instances (0/40)”.



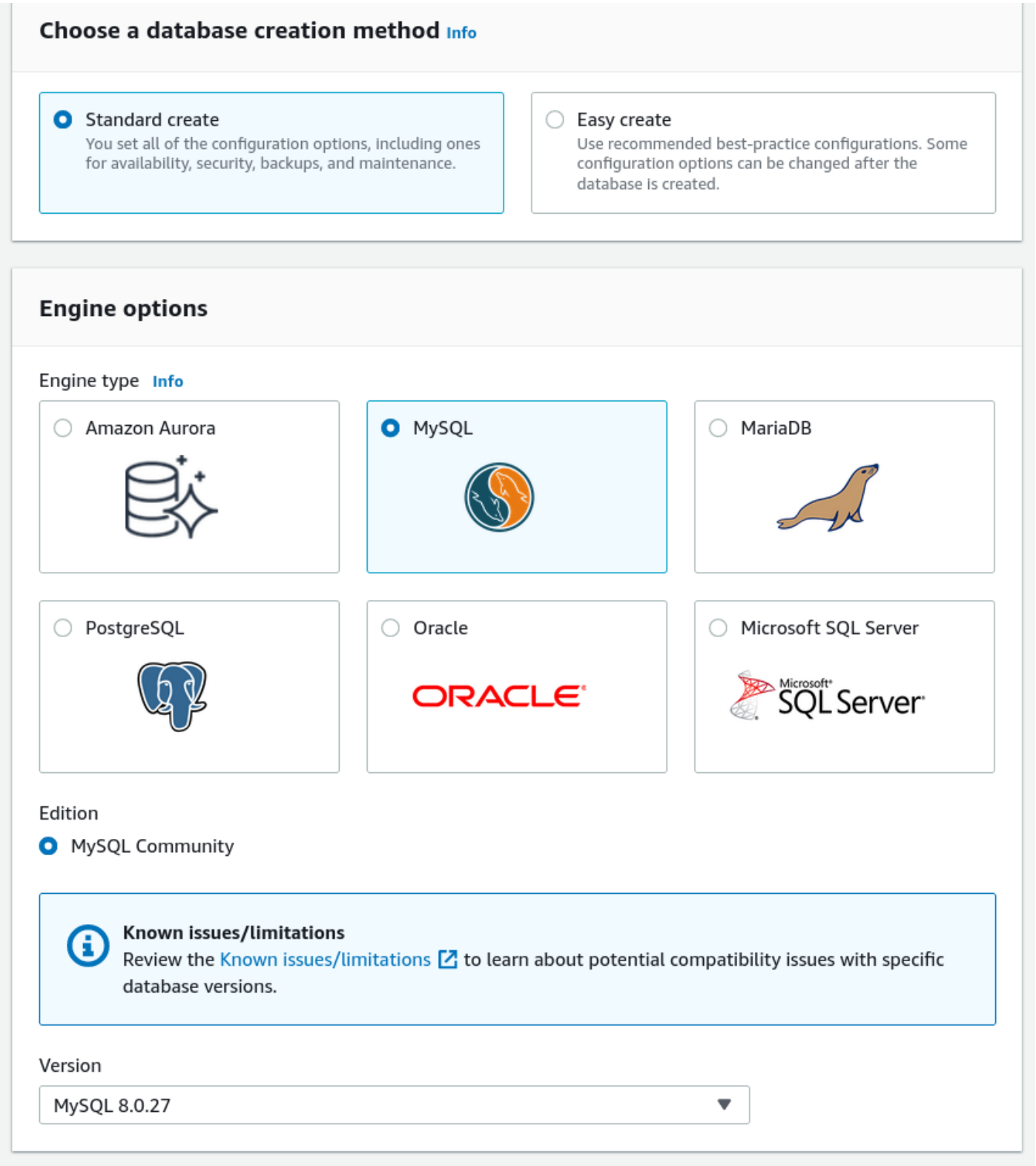
This page should appear familar as it's very similar to the AWS EC2 instance page. Let us create a new database by hitting the "Create Database" button.



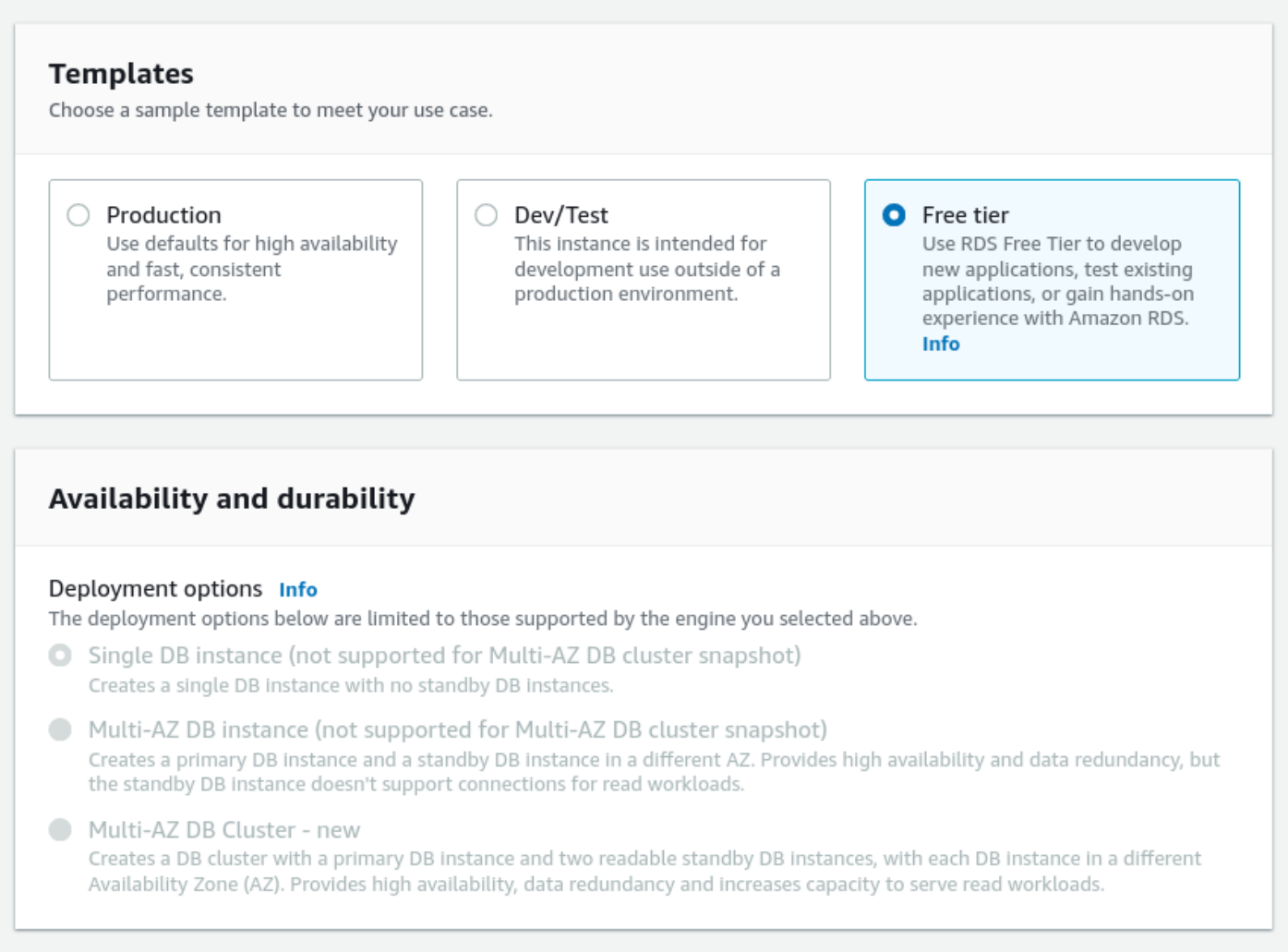
Warning

In the next section we cannot use the Easy Create method as it tries to create a IAM account which is not allowed in the labs. Going forward we would typically do this using Terraform so we can easily avoid these restrictions.

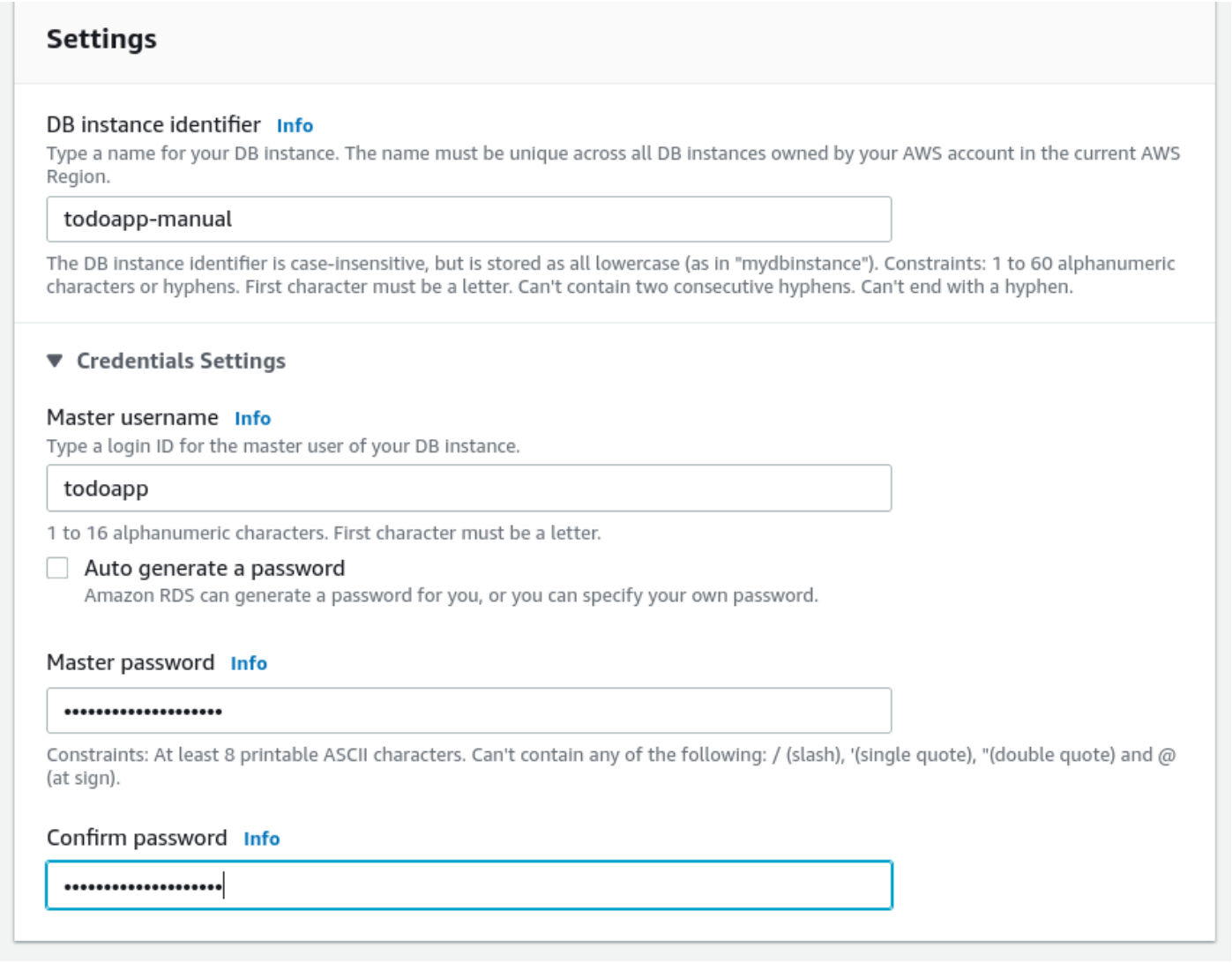
We will be creating a standard database so select standard and Postgresql. We will use version 14 which is a fairly recent release.



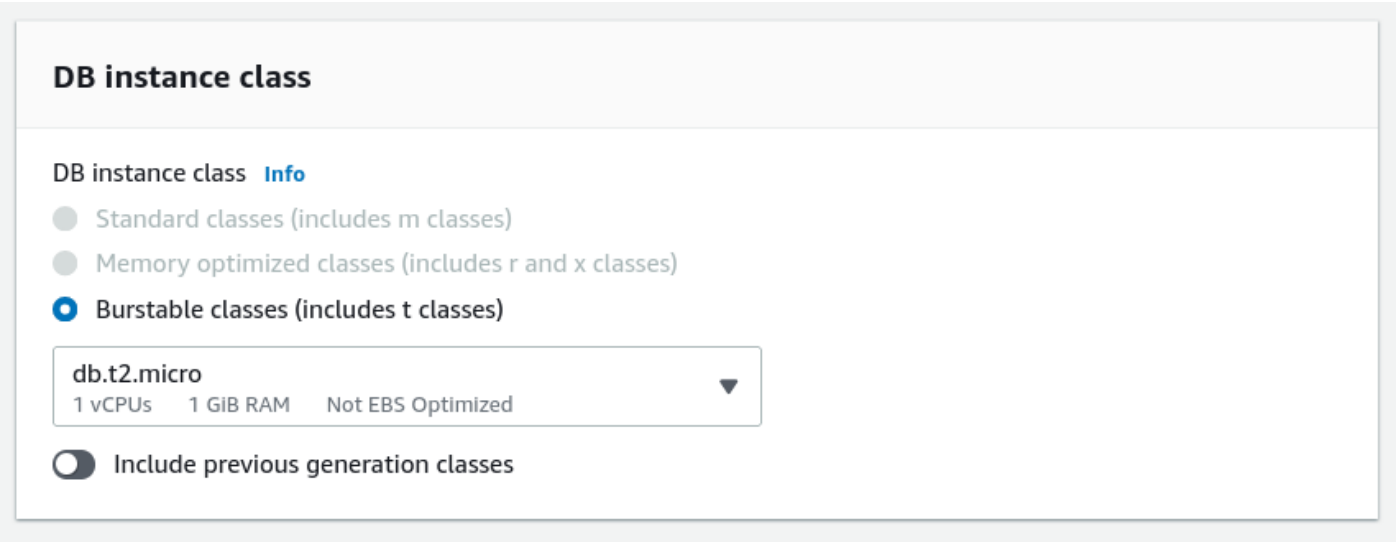
For today we are going to use “Free Tier” but in the future, you may wish to explore the different deployment options. Please peruse the available different options.



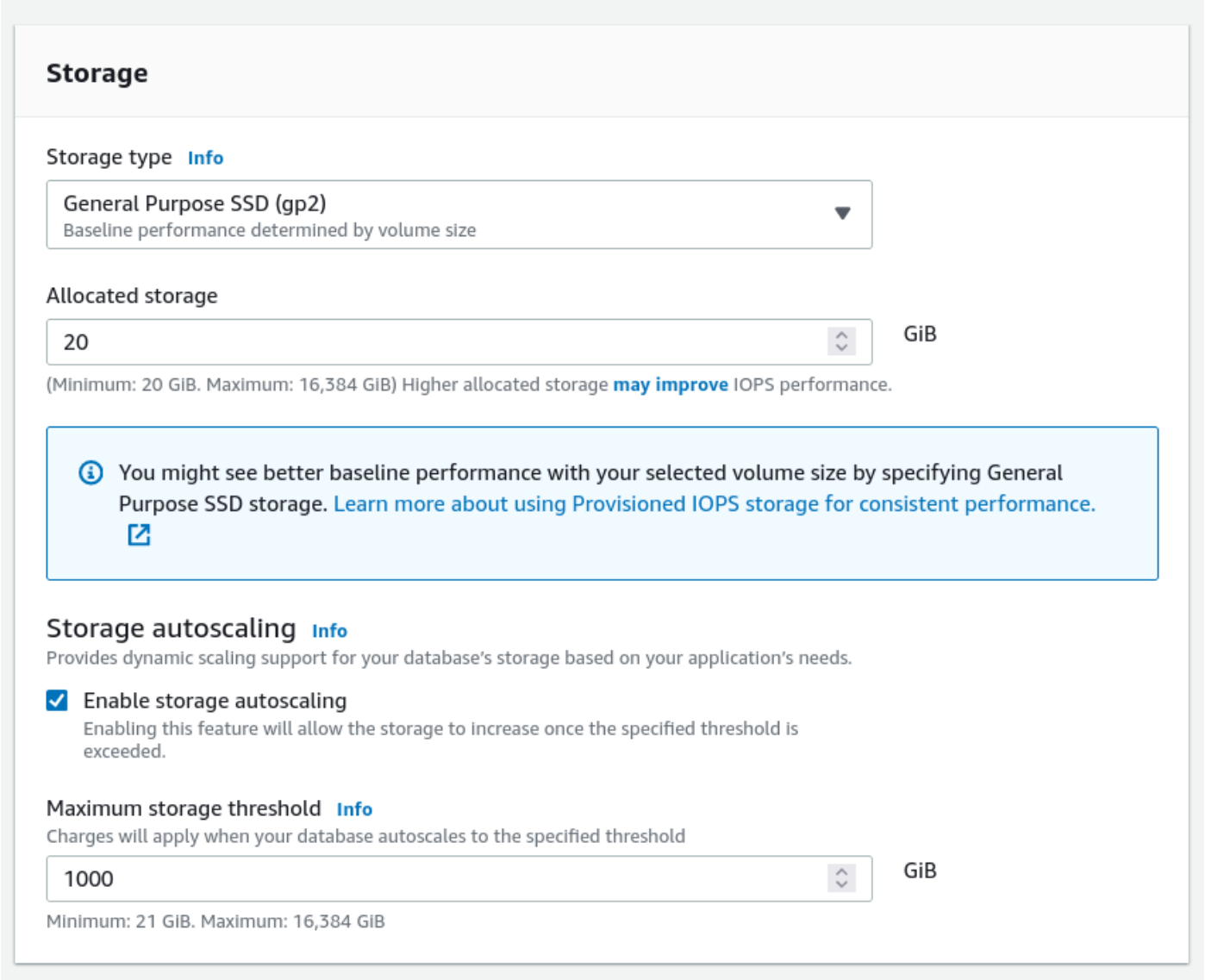
Now we need to name our database and create credentials to connect via. Here is where you can enter in credentials for the main account of the database.



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

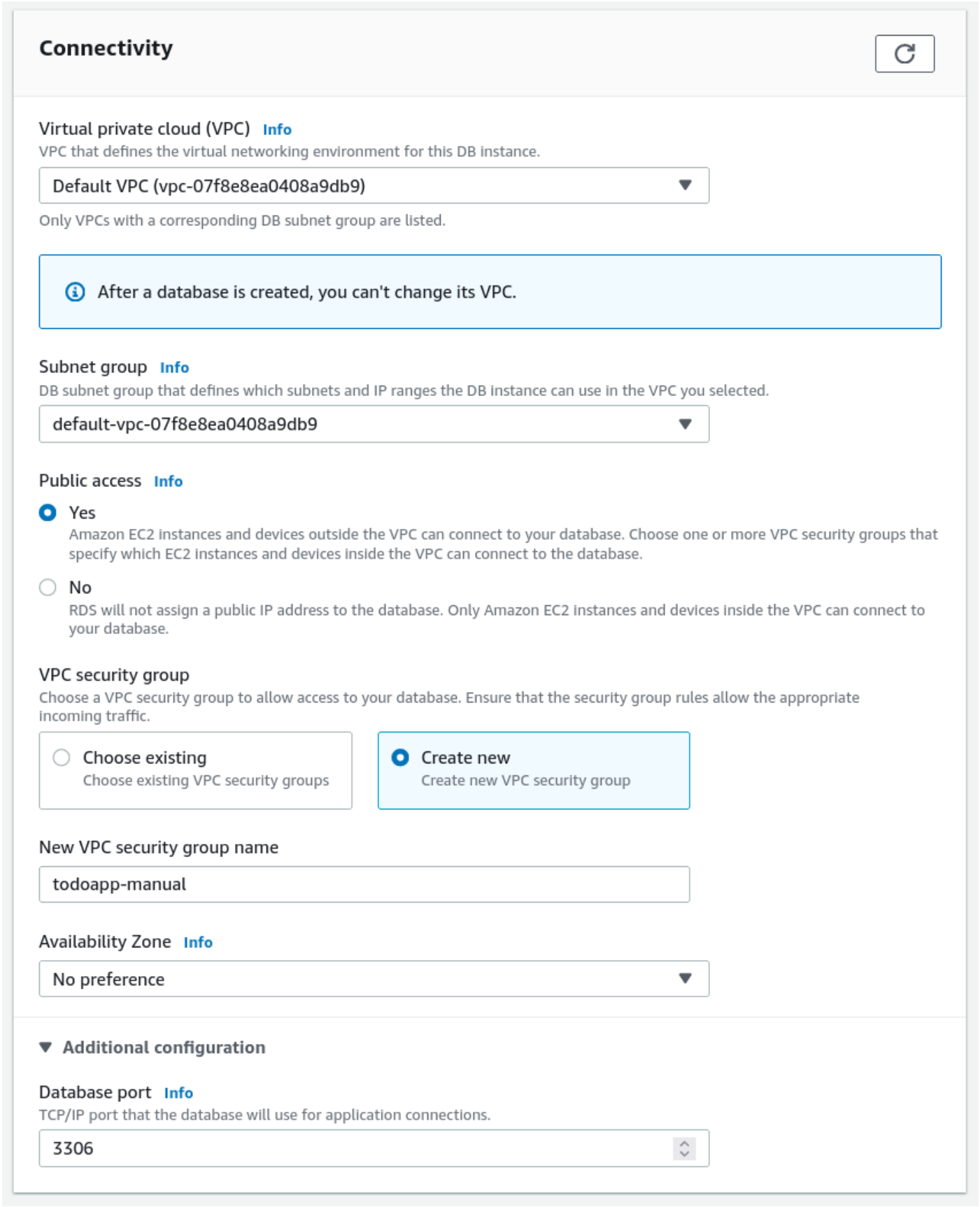


For storage we will leave all the default options.

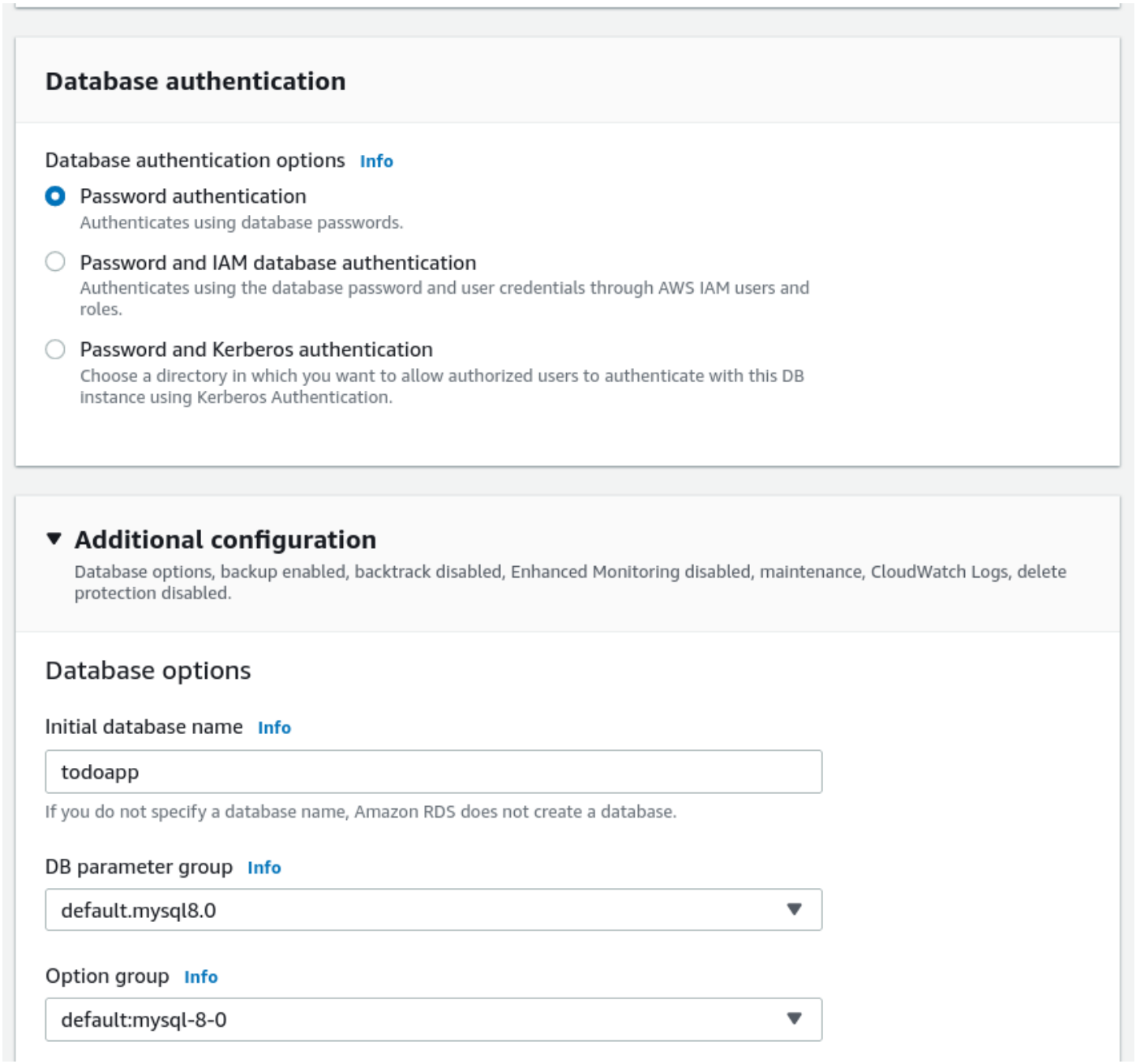


In connectivity we need to make sure our instance is publicly available. Usually you don't want to expose your databases publicly and, would instead, have a web server sitting in-front. For our learning purposes though we are gonna expose it directly just like we did with our EC2 instances early in the course.

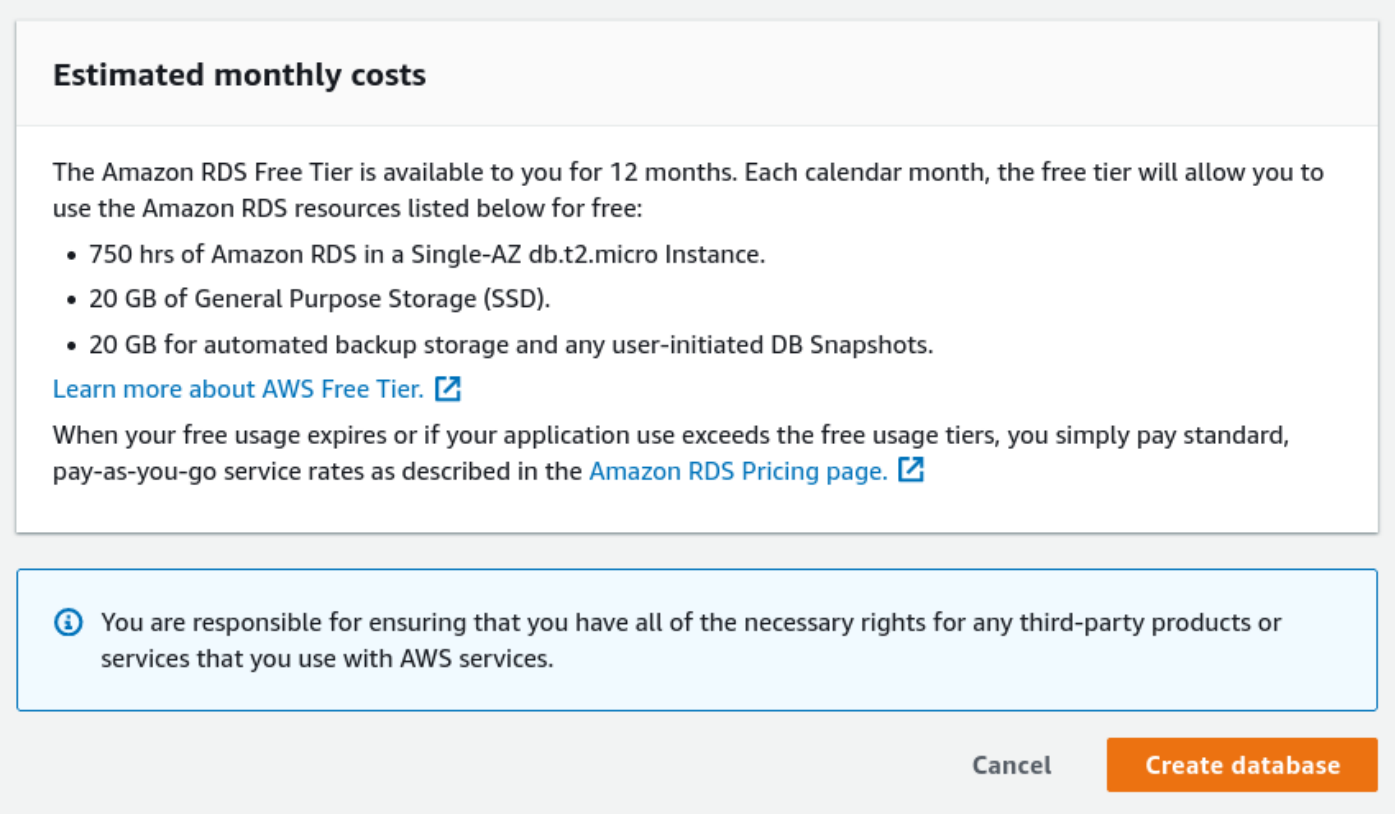
When selecting public access as yes we have to create a new Security Group, give this Security Group a sensible name.



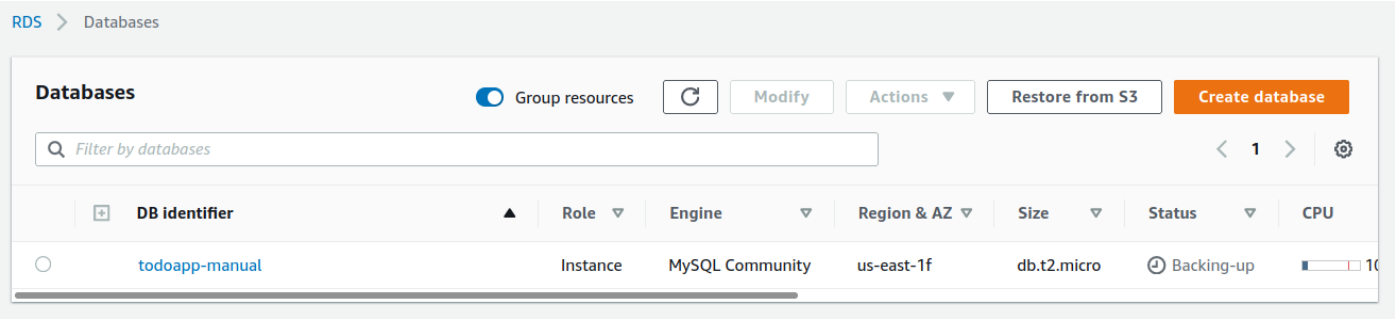
We will leave the authentication as password based but we need to expand the “Additional configuration”. Fill in the “Initial Database Name” section to be “todo”, this is similar to what we had in the Docker Composes environment variable.



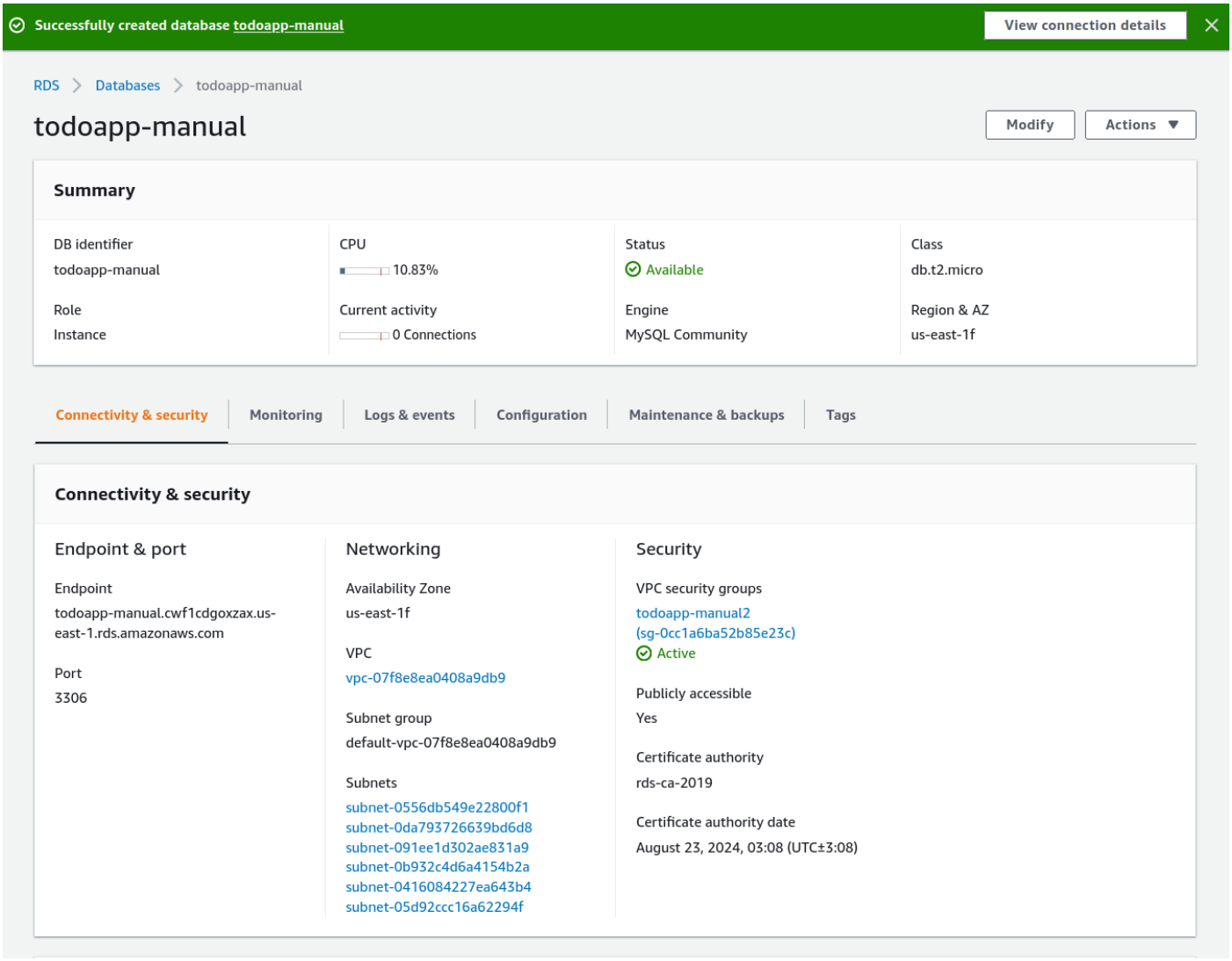
Now we can click create which will take some time.



Depending on your database it may take 10 to 30minutes to create, the larger and more complicated the setup the longer it usually takes. The database will also do a initial backup when its created.



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 copy into our docker compose file.



5 RDS Database with Terraform

Now would be a good time to browse the documentation for the RDS database in Terraform: 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:

```
   » cat main.tf
locals {
  password = "foobarbaz" # this is bad
}

resource "aws_db_instance" "todoapp-database" {
  allocated_storage = 20
  max_allocated_storage = 1000
  engine = "postgres"
  engine_version = "14"
  instance_class = "db.t4g.micro"
  db_name = "todo"
  username = "todo"
```

```
  password = local.password
  parameter_group_name = "default.postgres14"
  skip_final_snapshot = true
  vpc_security_group_ids = [aws_security_group.todo-database.id]
  publicly_accessible = true

  tags = {
    Name = "todo-database"
  }
}
```

Remember to create an appropriate security group as we did through the user interface.

```
   » cat main.tf
resource "aws_security_group" "todo-database" {
  name = "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 = "todo-database"
  }
}
```

TODO: Connect to the database and explore around.

6 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 maunally via EC2 so you can use the method you feel most confortable with.

For this practical we have made available a docker container running the todo application which you can use to deploy to AWS. 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.

6.1 Setup

All of the differnt ways that we can deploy our application we have already decided that we are gonna offload the database to AWS. This means that we can move all the "state" of our application away from our containerised environment.

To start off we are gonna use what we had above to create a database and the default terraform. Edit your files so that they match whats provided below:

```
» cat main.tf
terraform {
  required_providers {
      aws = {
          source = "hashicorp/aws"
          version = "~> 4.0"
      }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_files = ["./credentials"]
  default_tags {
      tags = {
          Course = "CSSE6400"
          Name = "TaskOverflow"
          Automation = "Terraform"
      }
  }
}

locals {
  image = "ghcr.io/csse6400/taskoverflow:latest"
  database_username = "administrator"
  database_password = "VerySecurePasswordByYourBoiEvan"
}

resource "aws_db_instance" "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.database.id]
  publicly_accessible = true
}

resource "aws_security_group" "database" {
  name = "todo-database"
  description = "Allow inbound Postgres 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"]
  }
}
```

The above sets up a RDS instances of postgres and a security group to allow access to it. We have also added a local variable to store the password for the database. Local variables in terraform can come through two ways, they can be in a variables block which can be overriden or they can be in a locals block which can be used to store values that are used in multiple places.

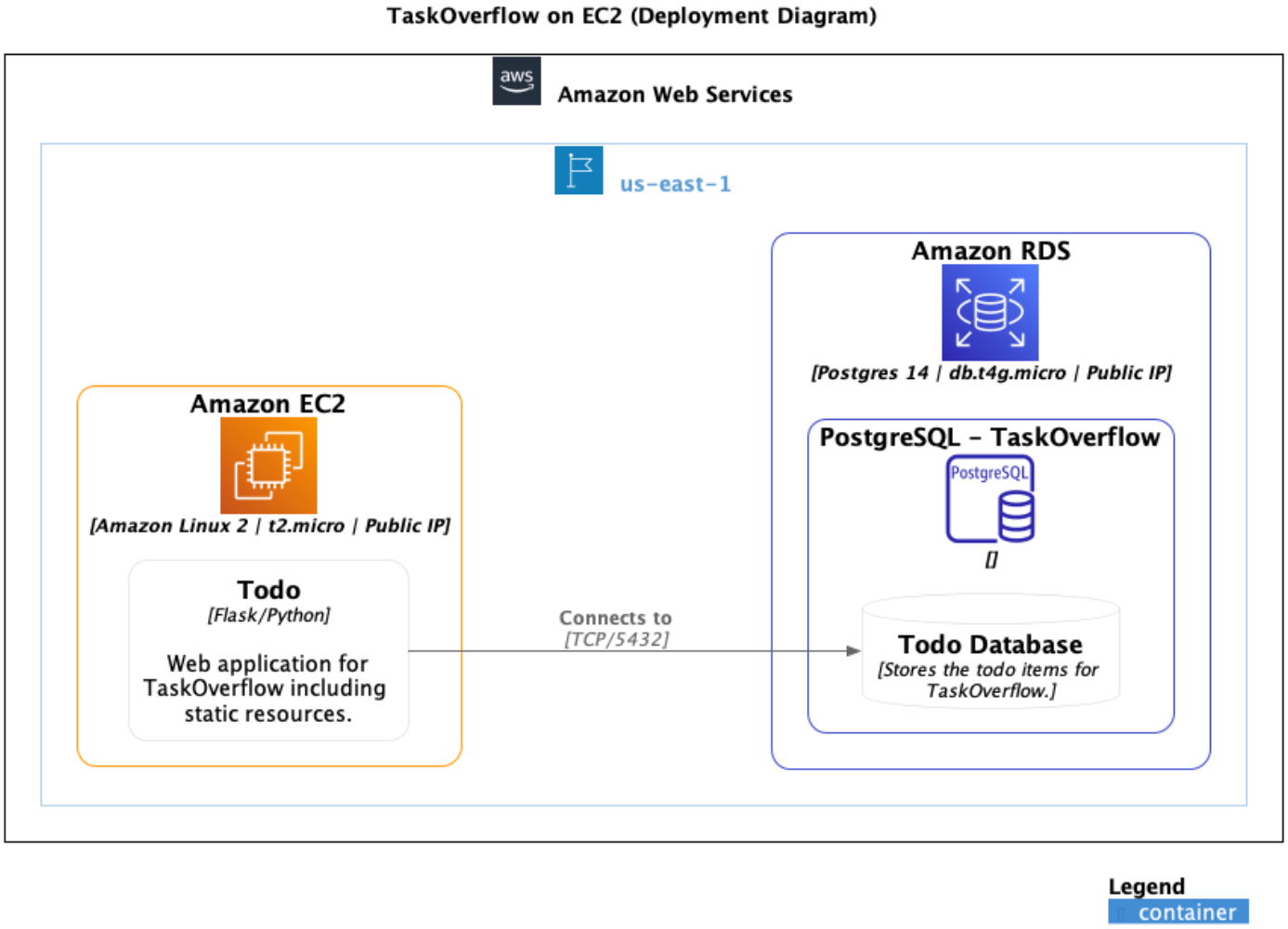
Now we can run terraform init and terraform apply to create our database. Once this is done we can move on to the next step. For the next step you must choose a path to follow, either EC2 or ECS, only choose one path for the practical, if you have trouble with one path you can always switch to the other. When switching you may want to destroy the resources you have created so far before starting the new path.

We would like to recommend that you start with the ECS path as it is the more modern way of deploying containers and is the path that we will be suggesting for the future but all tasks are also doable using EC2.

6.2 [Path A] EC2

Aside

This is not the recommended path but is provided for those who would like to use EC2 instead of ECS. This is also the path that was used in the first run of the course (2022).



Congrats you have chosen to go down the EC2 path which builds on what we experienced in the previouse practicals. To deploy our application we first need to get the EC2 instance running and install docker on the instance. We can do this by using the following terraform:

```
  » cat main.tf
resource "aws_instance" "todo" {
  ami = "ami-005f9685cb30f234b" // Amazon Linux 2
  instance_type = "t2.micro"
  key_name = "vockey" // allows us to SSH into the instance using the preconfigured
      key

  user_data_replace_on_change = true // If we change user_data this will force the
     box to be recreated
  user_data = <<-EOT
#!/bin/bash
yum update -y
  EOT

  security_groups = [aws_security_group.todo.name] // Firewall for the instance
}
```

Compared to the last time we used EC2 instances we have moved the `user_data` from a file to being defined inline and we have made sure to provide the `key_name` encase we want to ssh into the deployed instance.

If we ran terraform apply now our instance wouldnt do anything interesting and we havnt defined our security group yet. For the `user_data` we need to install docker and run the container that we have already built. We can do this by changing our `user_data`:

```
» cat main.tf
user_data = <<-EOT
#!/bin/bash
yum update -y
yum install -y docker
service docker start
systemctl enable docker
usermod -a -G docker ec2-user
docker run --restart always -e SQLALCHEMY_DATABASE_URI=postgresql://${local.
    database_username}:${local.database_password}@${aws_db_instance.database.address
    }:${aws_db_instance.database.port}/${aws_db_instance.database.db_name} -p
    6400:6400 ${local.image}
EOT
```

The first lines install docker and start the docker service and allow the ec2-user to be able to run docker commands. The last line is where the magic is happening but running the container by the docker cli. To get a refresher about the docker cli please see the lectures and the docker website.

Aside

The `-restart always` flag tells docker to restart the container if it crashes or is stopped. This is useful for our application as it may take a while for the database to be provisioned and we dont want to have to manually restart the container once the database is ready.

Now that our instance is ready to go we just need to make sure that its accessible from the internet. We can do this by creating a security group that allows traffic on port 6400 and attaching it to our instance.

```
» cat main.tf
resource "aws_security_group" "todo" {
  name = "todo"
  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"]
  }
}
```

Now we can run terraform init and terraform apply to deploy our entire application. Once this is done we can visit the public IP of our instance on port 6400 to see our application running.

```
output "url" {
  value = "http://${aws_instance.todo.public_ip}:6400/"
}
```

You should be presented with a very lightweight todo based application called TaskOverflow.

6.2.1 Finished Terraform

The below is the full terraform we built in this path.

```
  » cat main.tf
terraform {
  required_providers {
      aws = {
          source = "hashicorp/aws"
          version = "~> 4.0"
      }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_files = ["./credentials"]
  default_tags {
      tags = {
          Course = "CSSE6400"
          Name = "TaskOverflow"
          Automation = "Terraform"
      }
  }
}
```

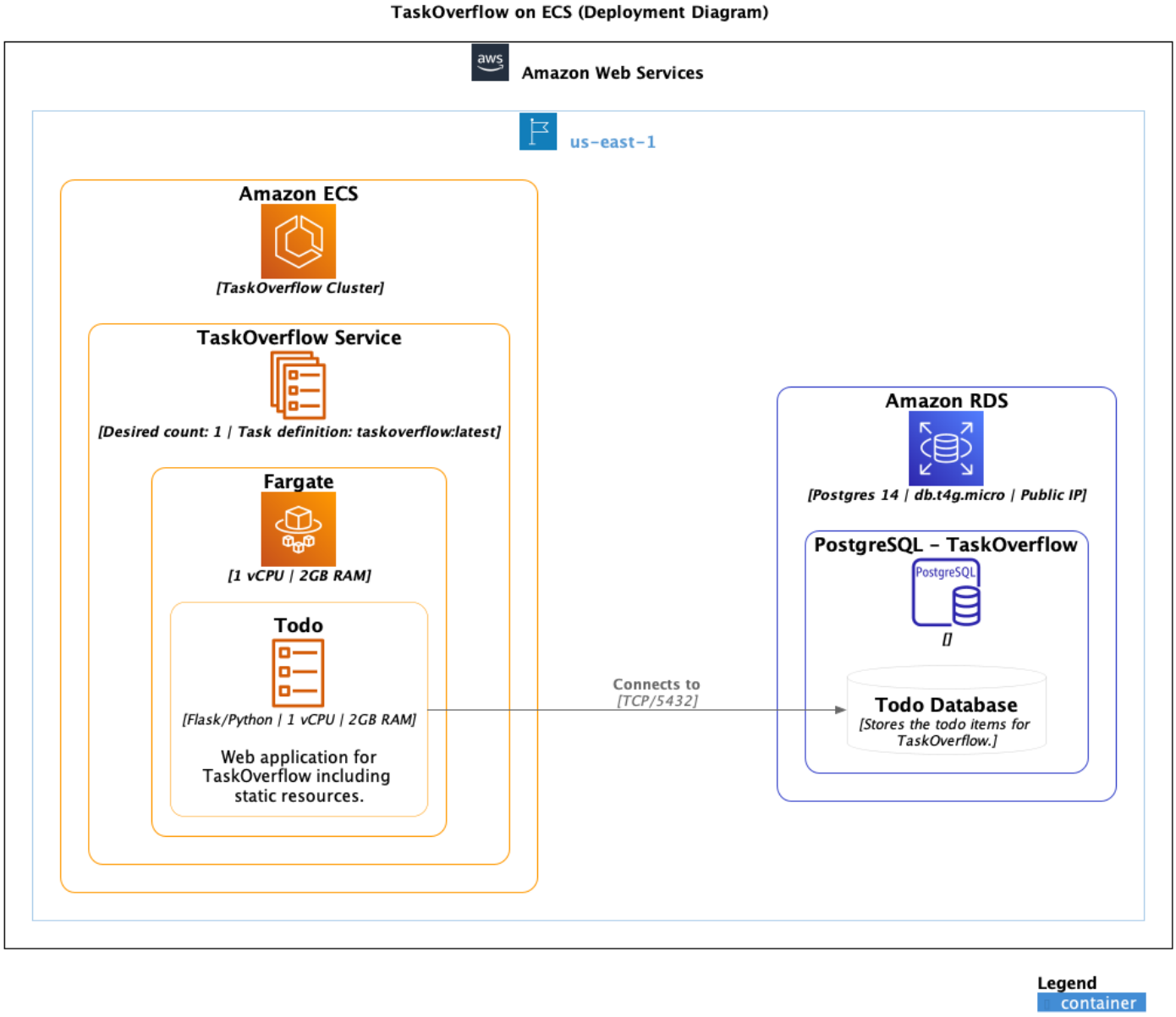
```
locals {
    image = "ghcr.io/csse6400/taskoverflow:latest"
    database_username = "administrator"
    database_password = "VerySecurePasswordByYourBoiEvan"
}

resource "aws_db_instance" "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.database.id]
  publicly_accessible = true
}

resource "aws_security_group" "database" {
  name = "todo-database"
  description = "Allow inbound Postgres 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"]
  }
}

resource "aws_instance" "todo" {
  ami = "ami-005f9685cb30f234b"
  instance_type = "t2.micro"
  key_name = "vockey"

  user_data_replace_on_change = true
  user_data = <<-EOT
#!/bin/bash
```

```
yum update -y
yum install -y docker
service docker start
systemctl enable docker
usermod -a -G docker ec2-user
docker run --restart always -e SQLALCHEMY_DATABASE_URI=postgresql://${local.
    database_username}:${local.database_password}@${aws_db_instance.database.address
    }:${aws_db_instance.database.port}/${aws_db_instance.database.db_name} -p
    6400:6400 ${local.image}
  EOT

  security_groups = [aws_security_group.todo.name]
}

resource "aws_security_group" "todo" {
  name = "todo"
  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"]
  }
}

output "url" {
  value = "http://${aws_instance.todo.public_ip}:6400/"
}
```

6.3 [Path B] ECS

Aside

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



Congrats you have choosen to go down the ECS path which mimics the same environment that you have via Docker Compose but as a service on AWS. This path is new for the course this year so please let your tutors know any particular issues you have.

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

```
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]
  }
}
```

The way in terraform that we can get external information into the environment is by using data sources. These are just like resources but they are not created or destroyed see the below to see the syntactual differences.

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

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

```
  » cat main.tf
resource "aws_ecs_task_definition" "todo" {
  family = "todo"
  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}
           @${aws_db_instance.database.address}:${aws_db_instance.database.port}/${
           aws_db_instance.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
}
```

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

```
 » cat main.tf
resource "aws_ecs_service" "taskoverflow" {
  name = "taskoverflow"
  cluster = aws_ecs_cluster.taskoverflow.id
  task_definition = aws_ecs_task_definition.todo.arn
  desired_count = 1
  launch_type = "FARGATE"

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

```
  » cat main.tf
resource "aws_security_group" "todo" {
  name = "todo"
  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"]
  }
}
```

6.3.1 Finished Terraform

```
  » cat main.tf
terraform {
  required_providers {
      aws = {
          source = "hashicorp/aws"
          version = "~> 4.0"
      }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_files = ["./credentials"]
  default_tags {
      tags = {
          Course = "CSSE6400"
```

```
            Name = "TaskOverflow"
            Automation = "Terraform"
        }
    }
}

locals {
    image = "ghcr.io/csse6400/taskoverflow:latest"
    database_username = "administrator"
    database_password = "VerySecurePasswordByYourBoiEvan"
}

resource "aws_db_instance" "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.database.id]
  publicly_accessible = true
}

resource "aws_security_group" "database" {
  name = "todo-database"
  description = "Allow inbound Postgres 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"]
  }
}

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

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

resource "aws_ecs_task_definition" "todo" {
  family = "todo"
  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.database.address}:${aws_db_instance.database.port}/${
           aws_db_instance.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
}

data "aws_vpc" "default" {
```

```
    default = true
}

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

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

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

resource "aws_security_group" "todo" {
  name = "todo"
  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"]
  }
}
```

6.4 [Path C] EKS / K8S

This path is not available in the course yet but we recommend that if you liked the course to have a look at Kubernetes as it is widly used in industry.

References

[1] B. Webb, "Infrastructure as code," March 2022. https://csse6400.uqcloud.net/handouts/iac.pdf.