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.

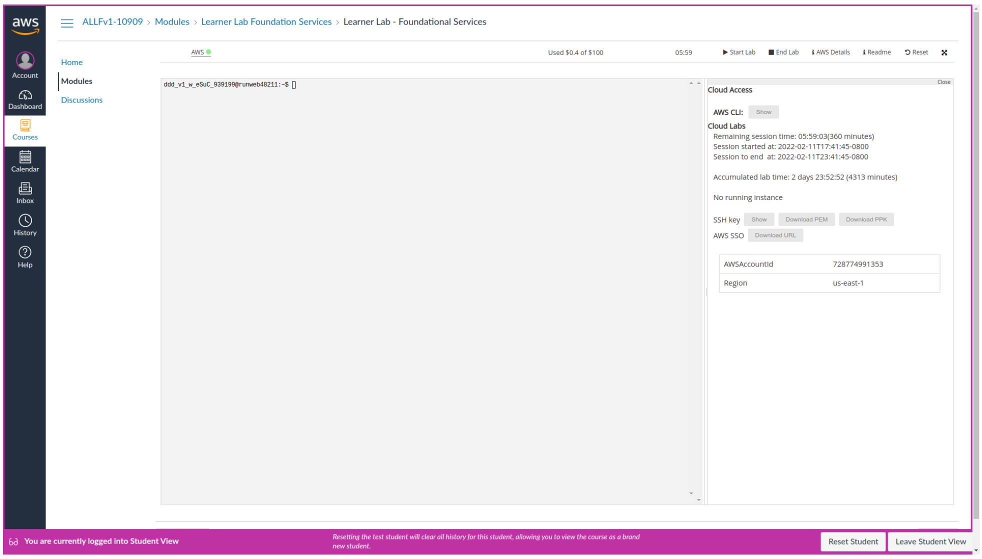
2 Terraform in AWS Learner Labs

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 CodeSpaces[1]

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 <u>not</u> share this file contents — do <u>not</u> 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"]
}
```

[1] If 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
```

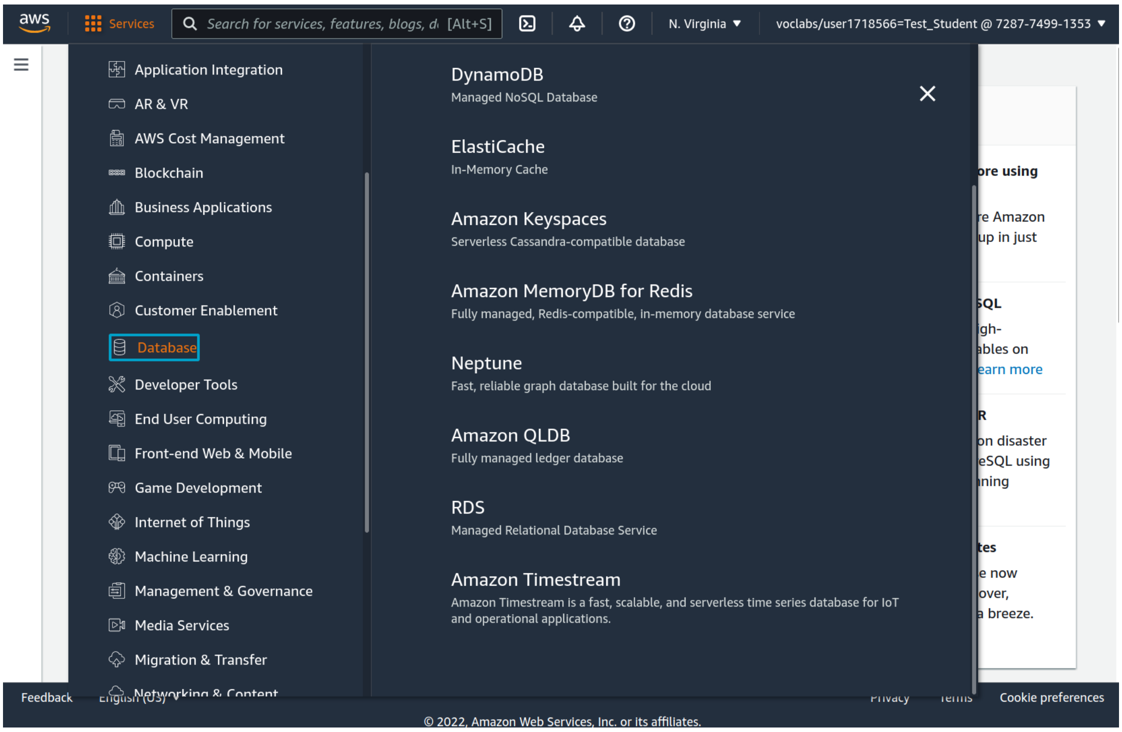
As we currently have no resources configured, it should find that no changes are required. Note that this does not ensure our credentials are correctly configured as Terraform has no reason to try authenticating yet.

3 Deploying a Database in AWS

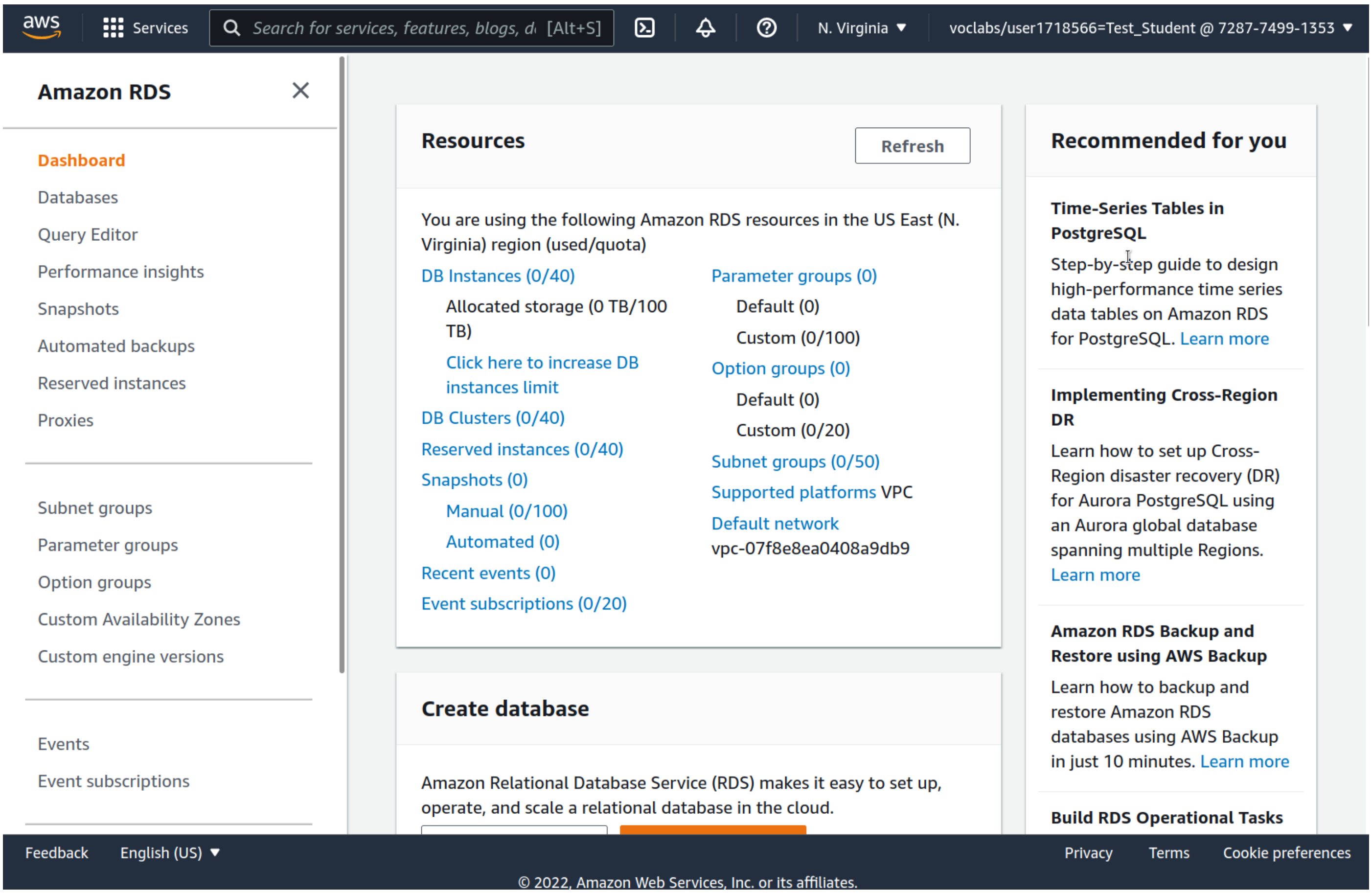
Warning

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.

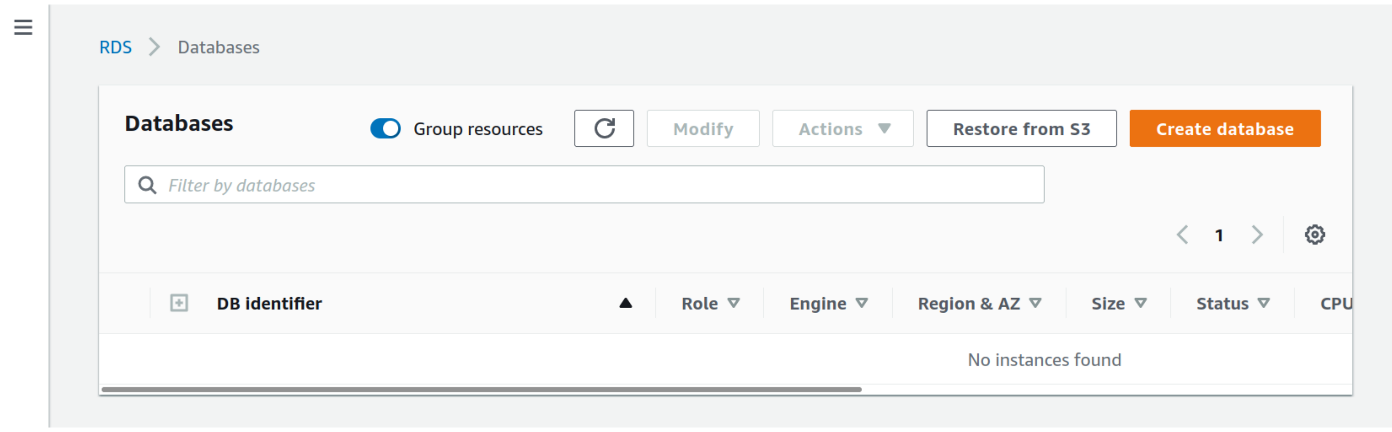
To get started let us jump into the lab environment and have a look at AWS RDS which is an AWS managed database service. To get to the RDS service either search it or browse Services -> Database -> RDS as shown below.



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.



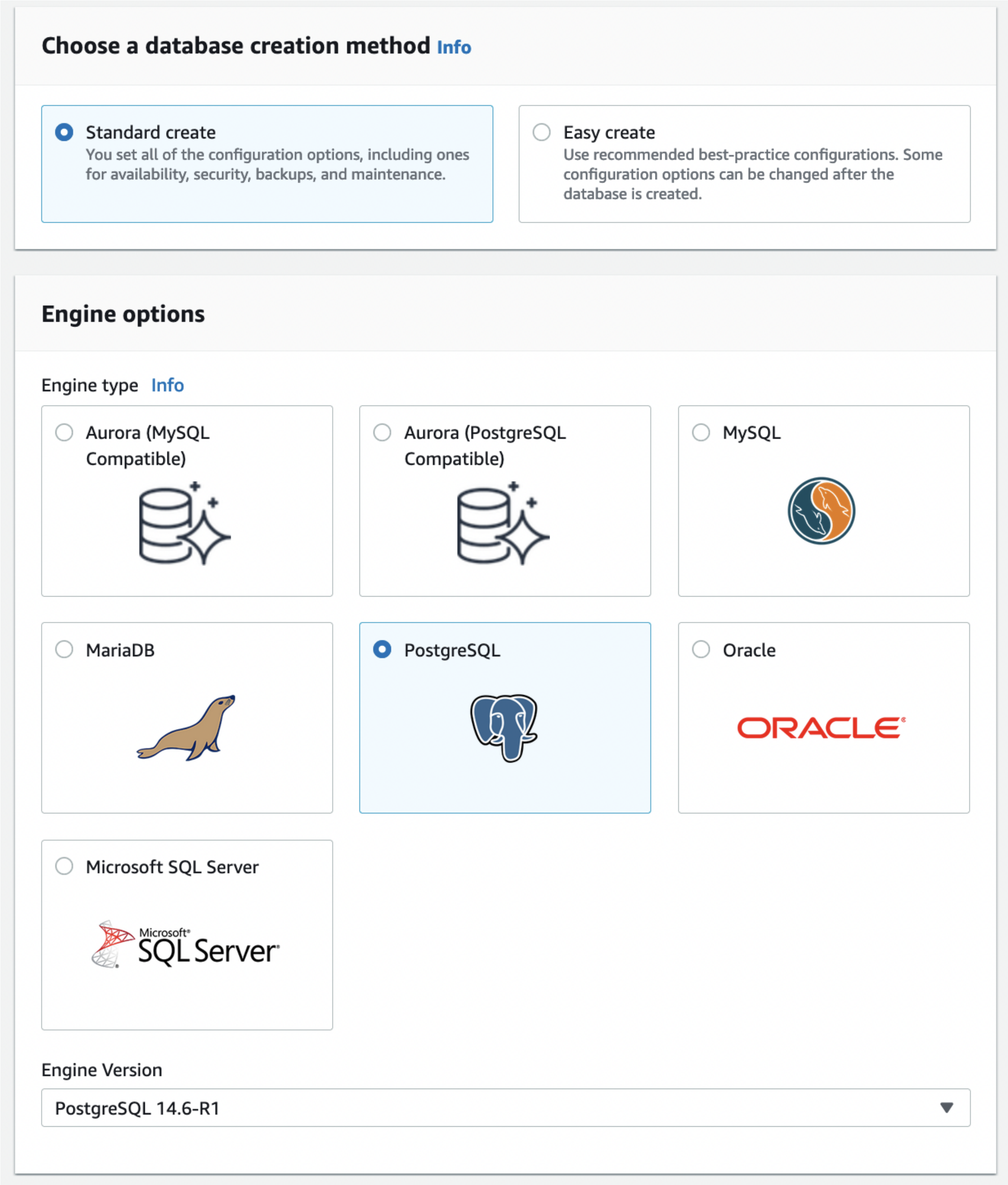
This page should appear familiar as it is very similar to the AWS EC2 instance page. Let us create a new database by hitting the “Create Database” button.



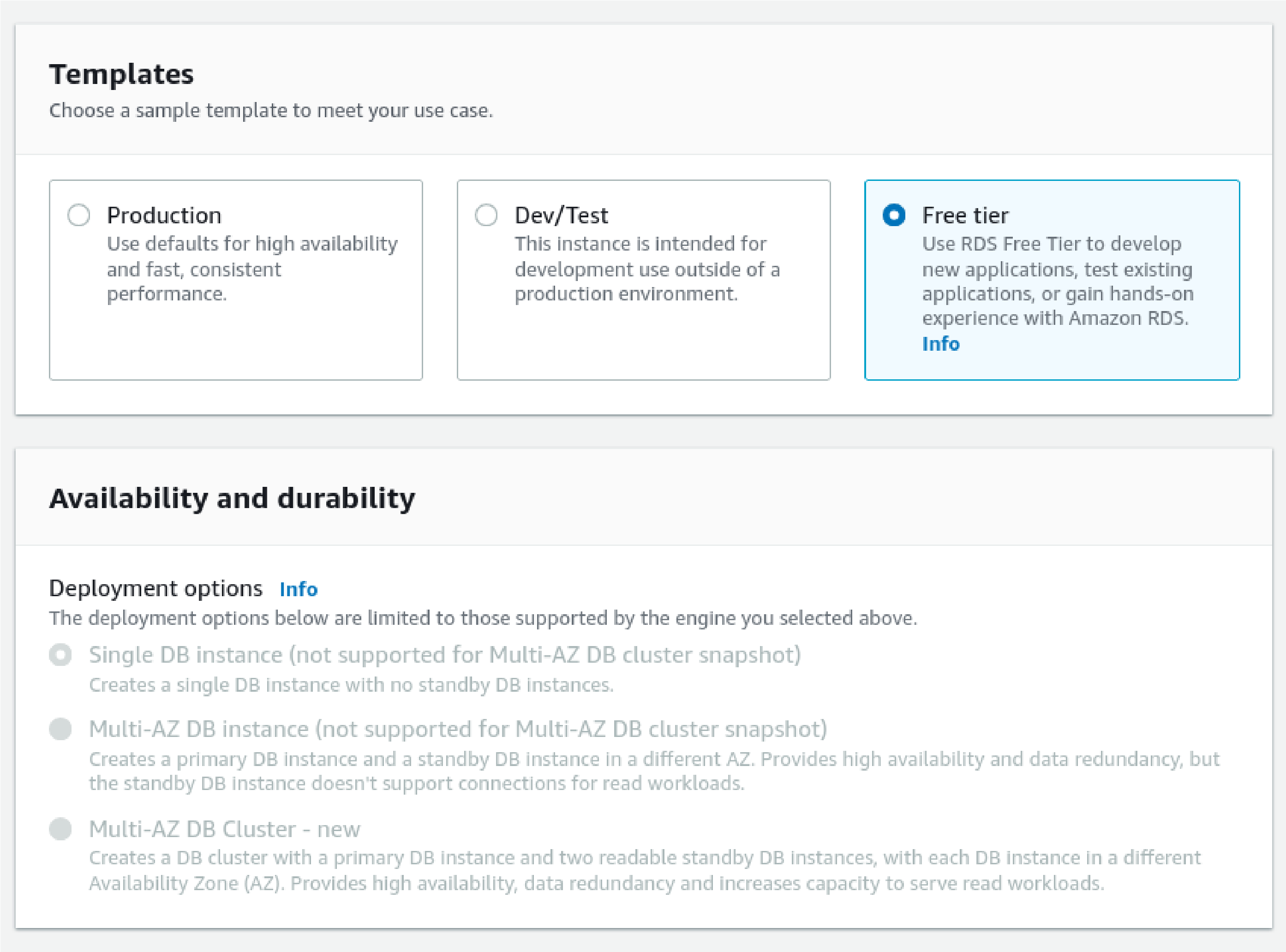
Warning

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.

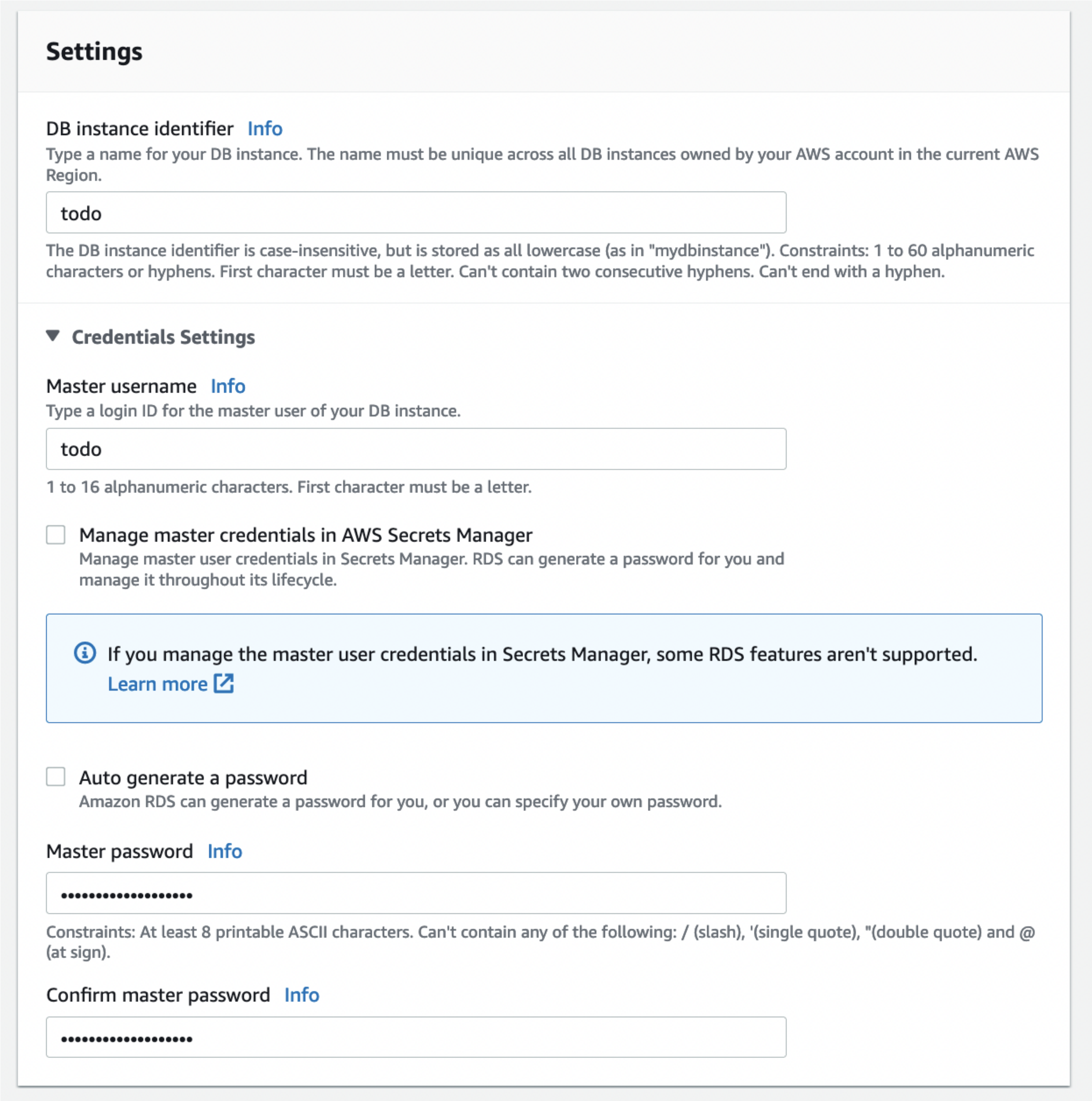
We will be creating a standard database so select standard and PostgreSQL. We will use version 14, which is a fairly recent release.



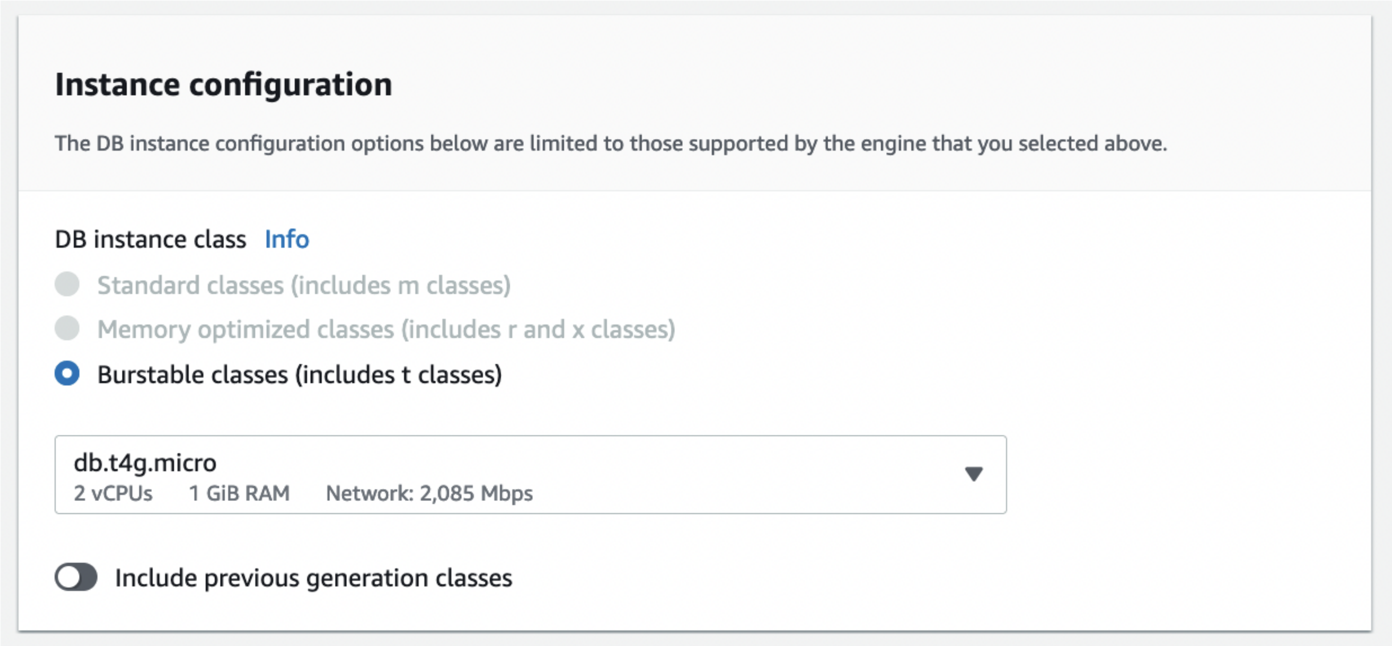
For today we are going to use "Free Tier" but in the future, you may wish to explore the different deployment options. Please peruse the available different options.



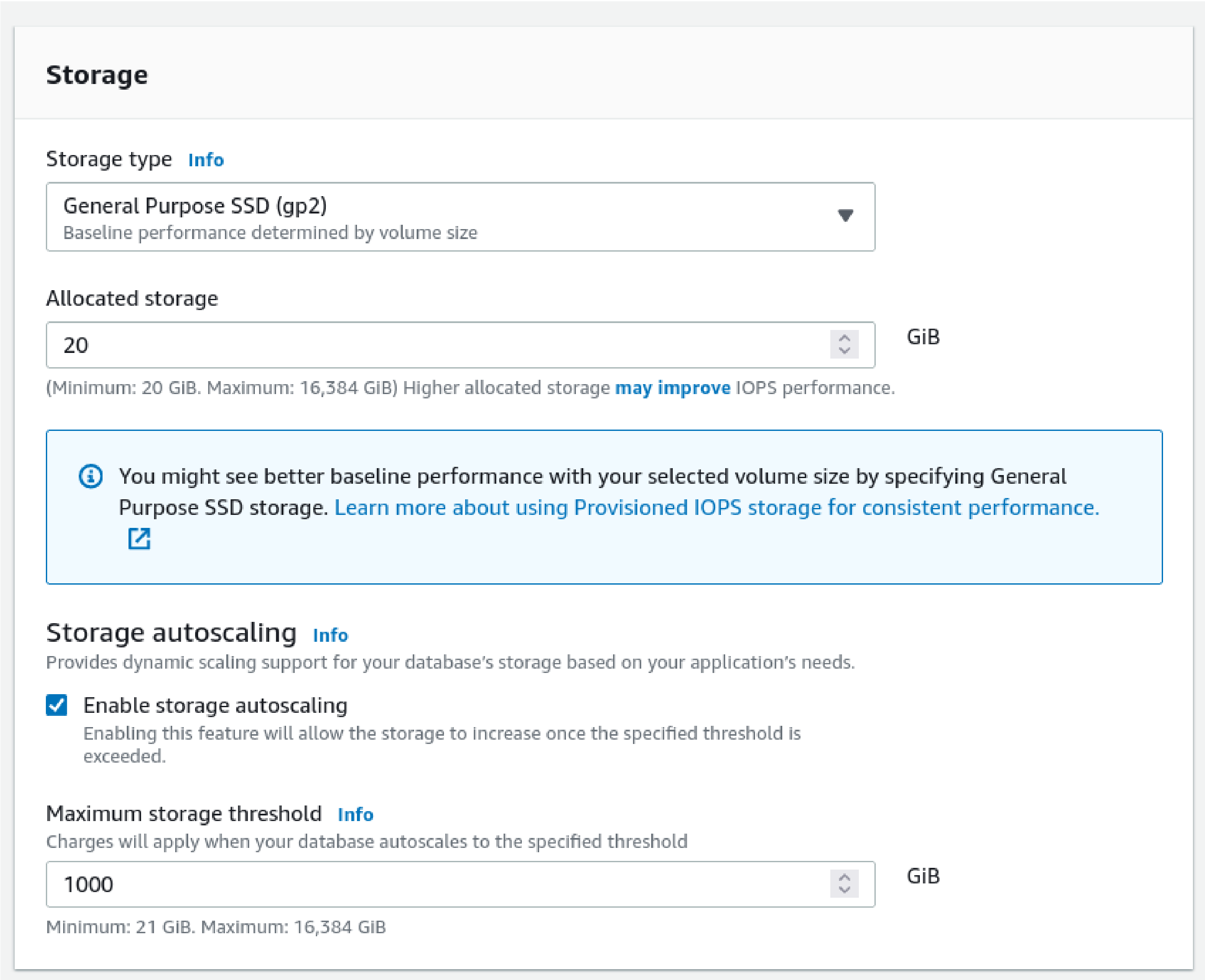
Now we need to name our database and create credentials to use when connecting from our application. Enter memorable credentials as these will be used later.



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

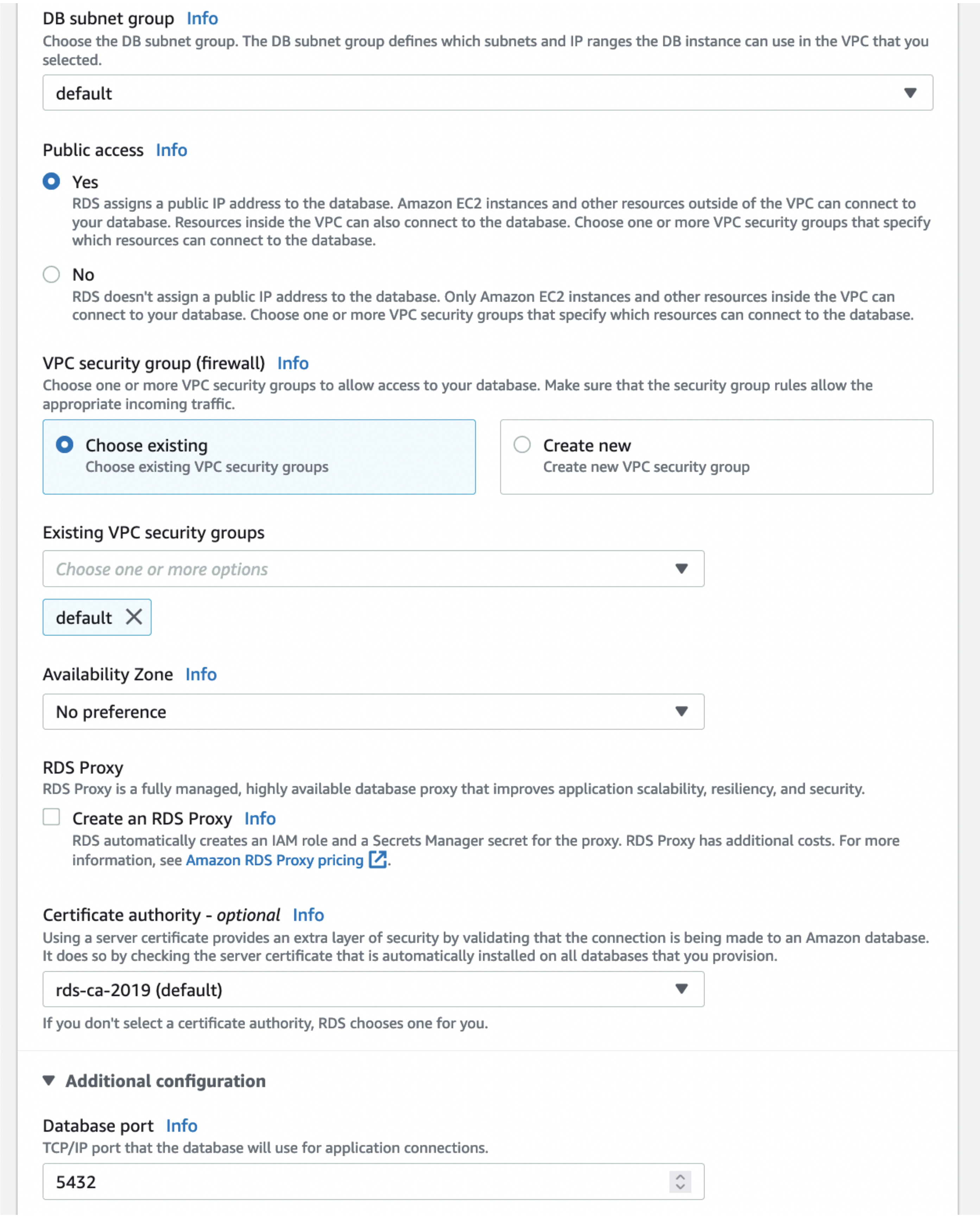


For storage we will leave all the default options.

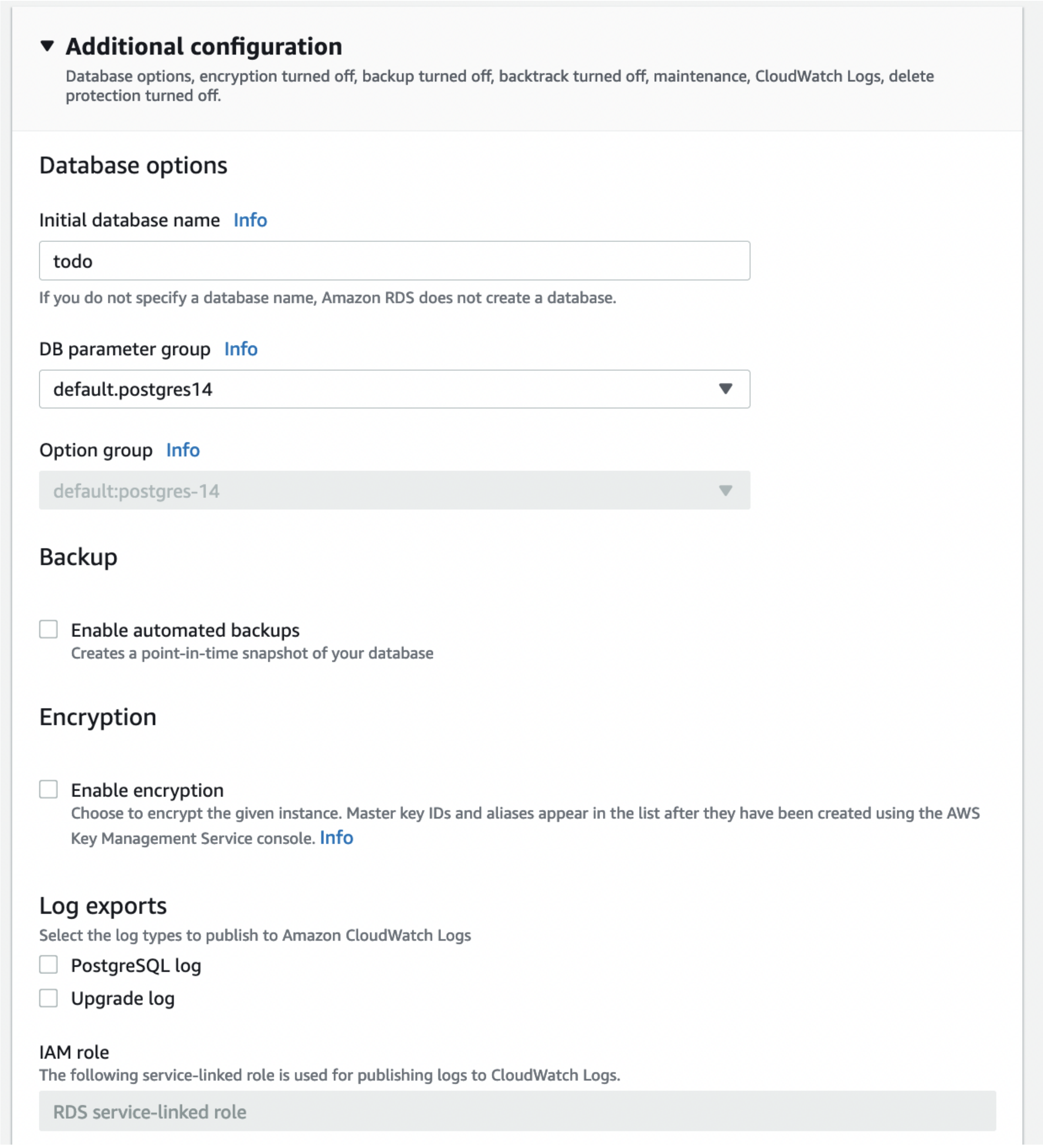


In connectivity we need to make sure our instance is publicly available. Usually you do not want to expose your databases publicly and, would instead, have a web server sitting in-front. For our learning purposes though we are going to expose it directly just like we did with our EC2 instances early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

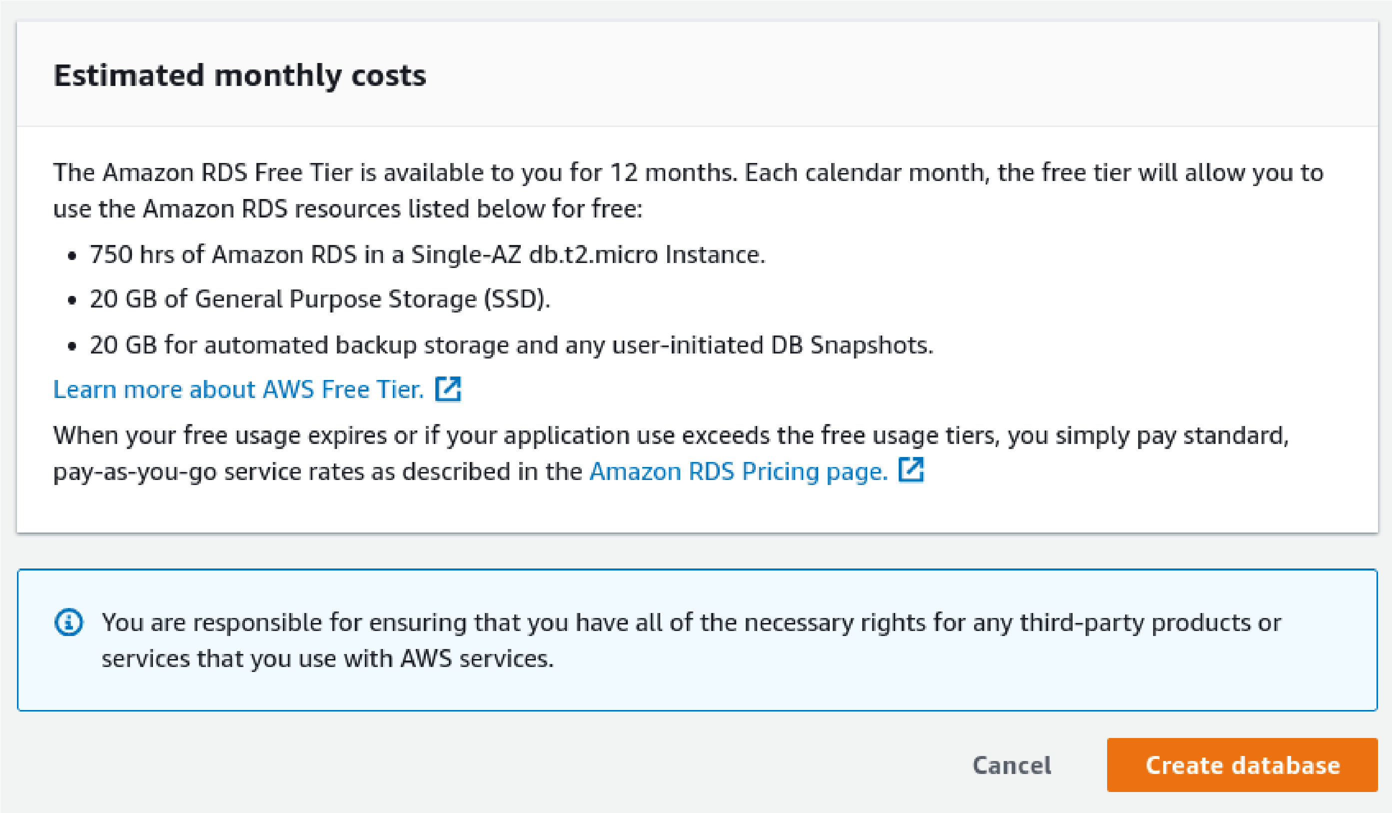
When selecting public access as yes we have to create a new Security Group, give this Security Group a sensible name.



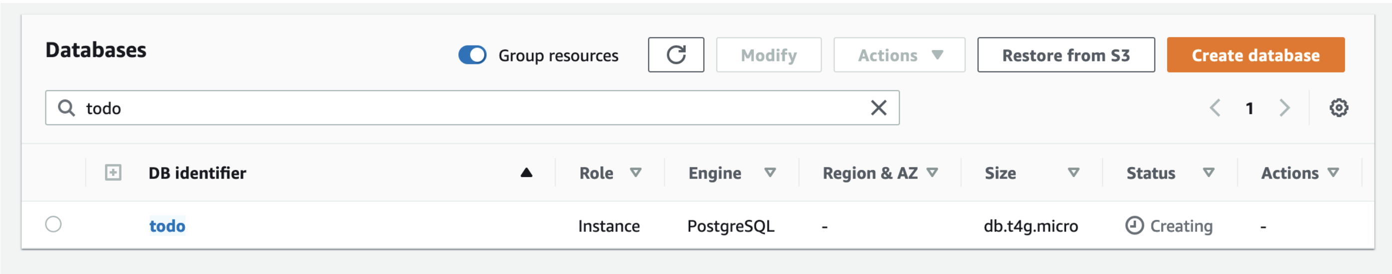
We will leave the authentication as password based but we need to expand the "Additional configuration". Fill in the "Initial Database Name" section to be "todo", this will automatically create the database that our todo application expects to connect to.



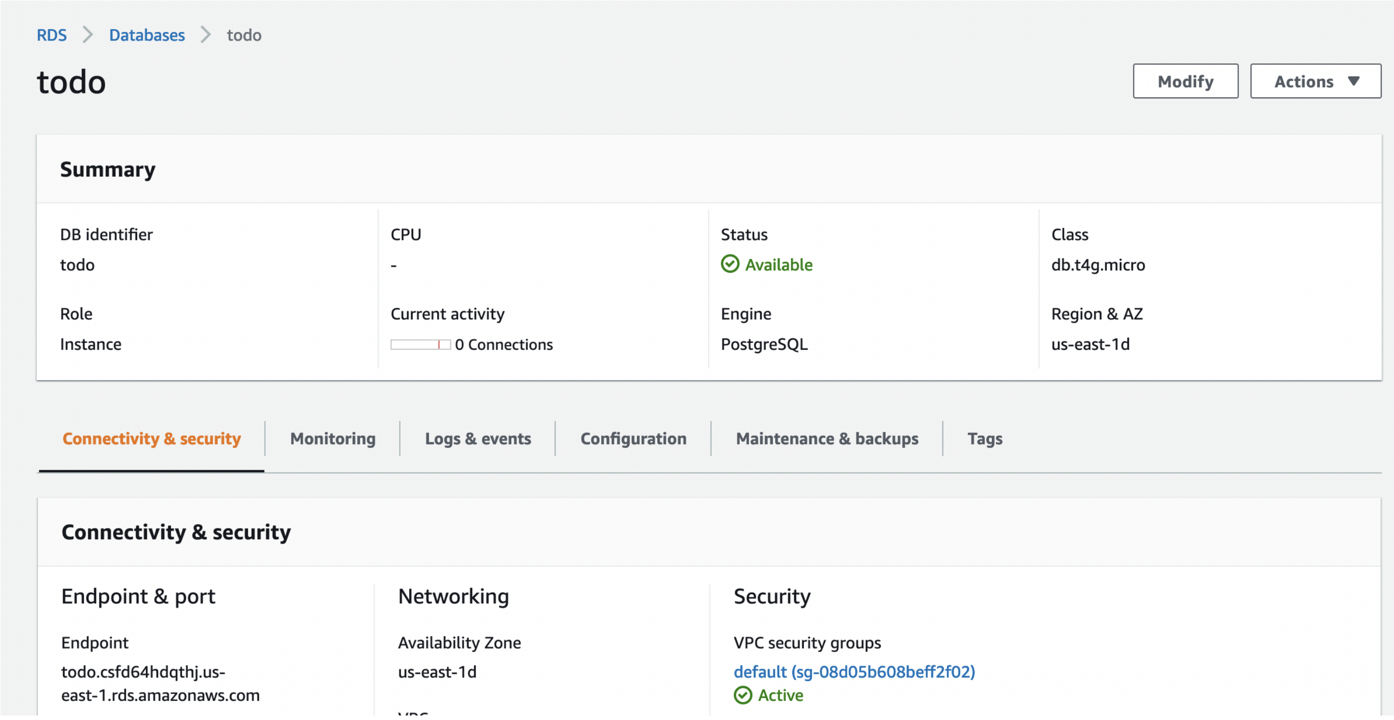
Now we can click create which will take some time.



It may take 10 to 30 minutes to create. 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 configure our TaskOverflow application to use.



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:

```
» cat main.tf
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_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 maunally 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]

[2] If 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.

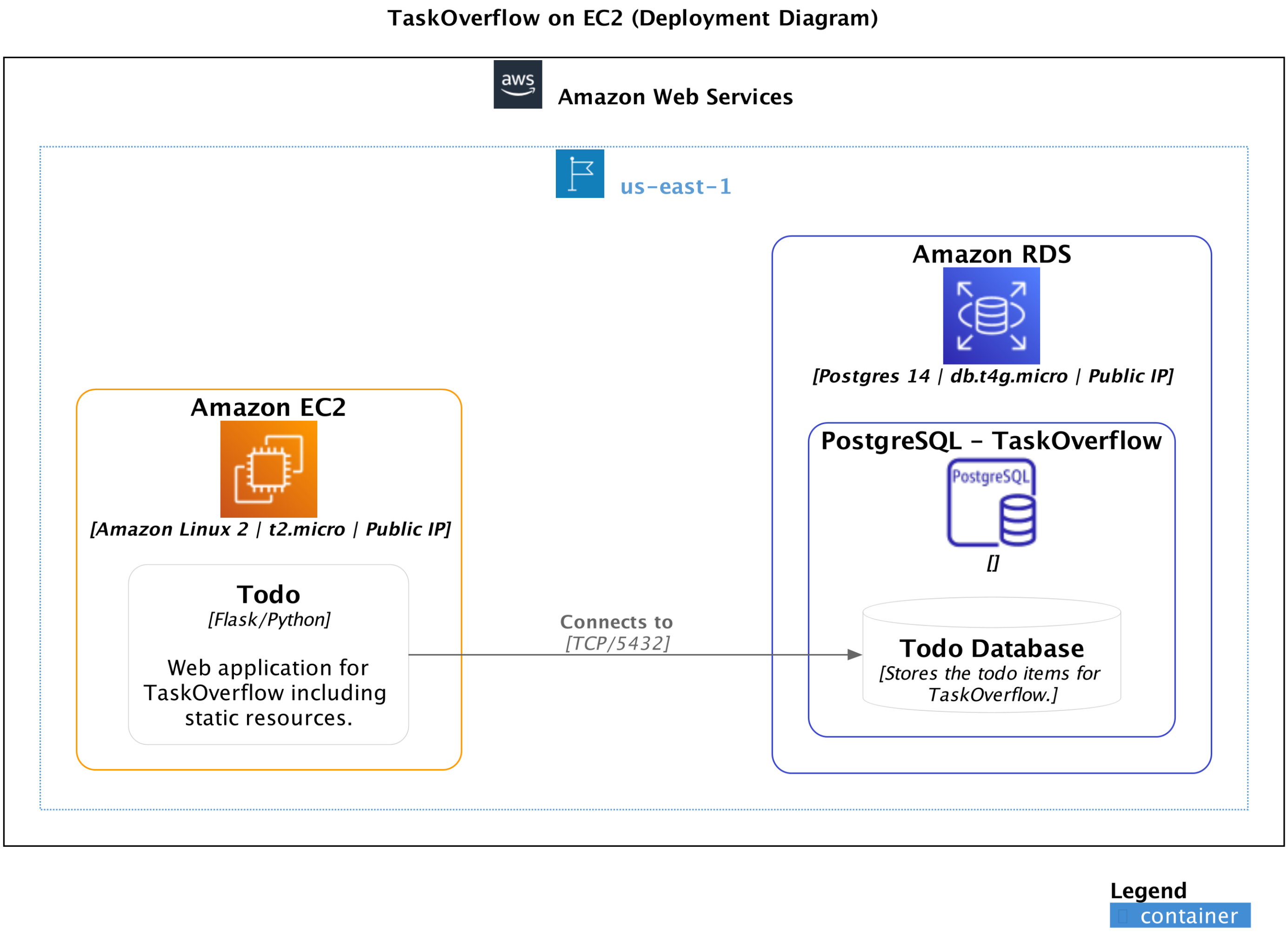
We have also added a local variable for us to use later. Variables in Terraform can be populated via two mechanisms, they can be in a variables block which can be overridden, or they can be in a locals block which can be used to store values that are used in multiple places.

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.

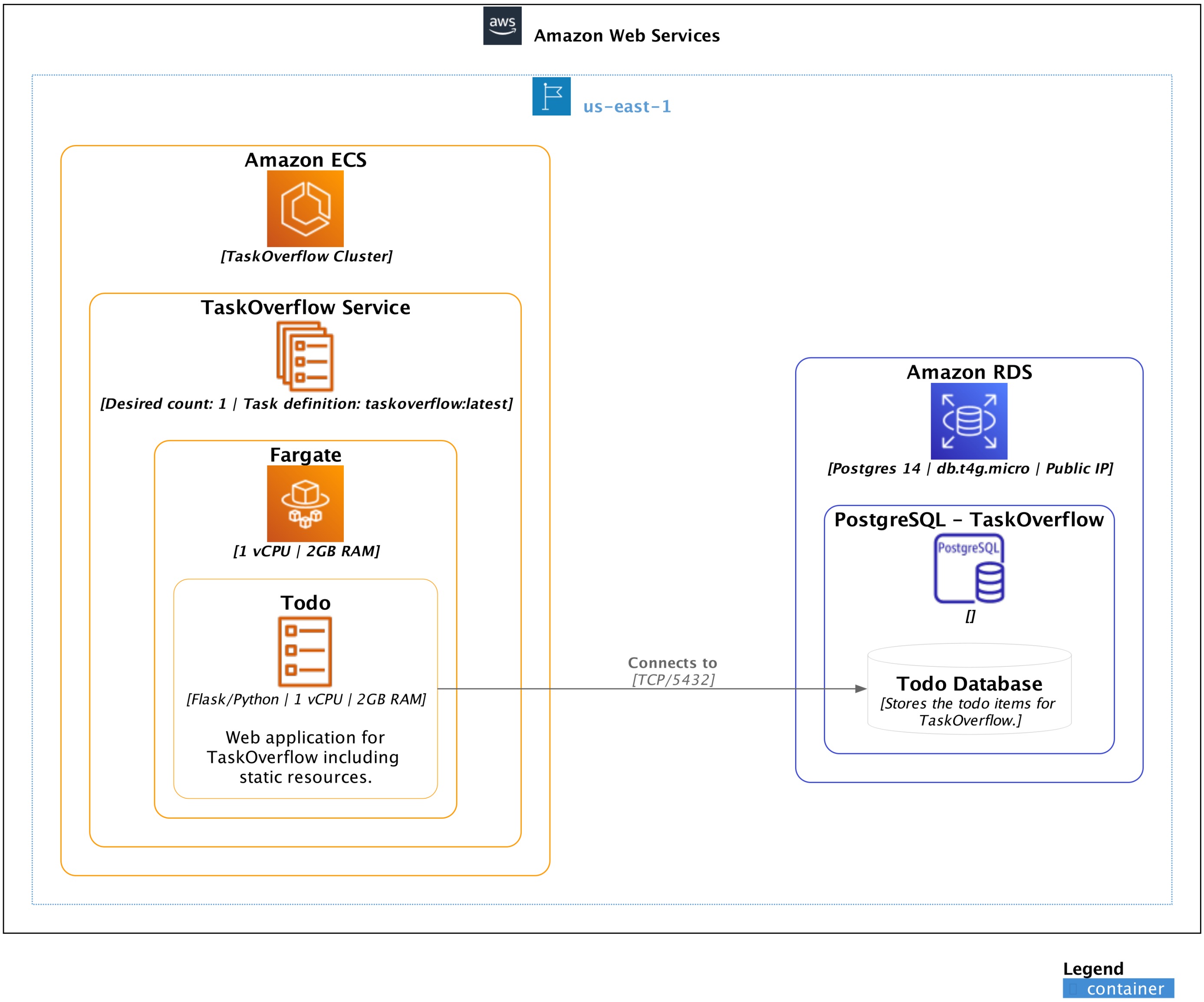


5.3 [Path B] ECS

Aside

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

TaskOverflow on ECS (Deployment Diagram)



Congratulations! You have choosen 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 it's 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 equivalen 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 Kubernetes[3] 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 being 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 contianer 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.