Storing Stuff // TODO

Software Architecture

March 21, 2022

Evan Hughes & Brae Webb

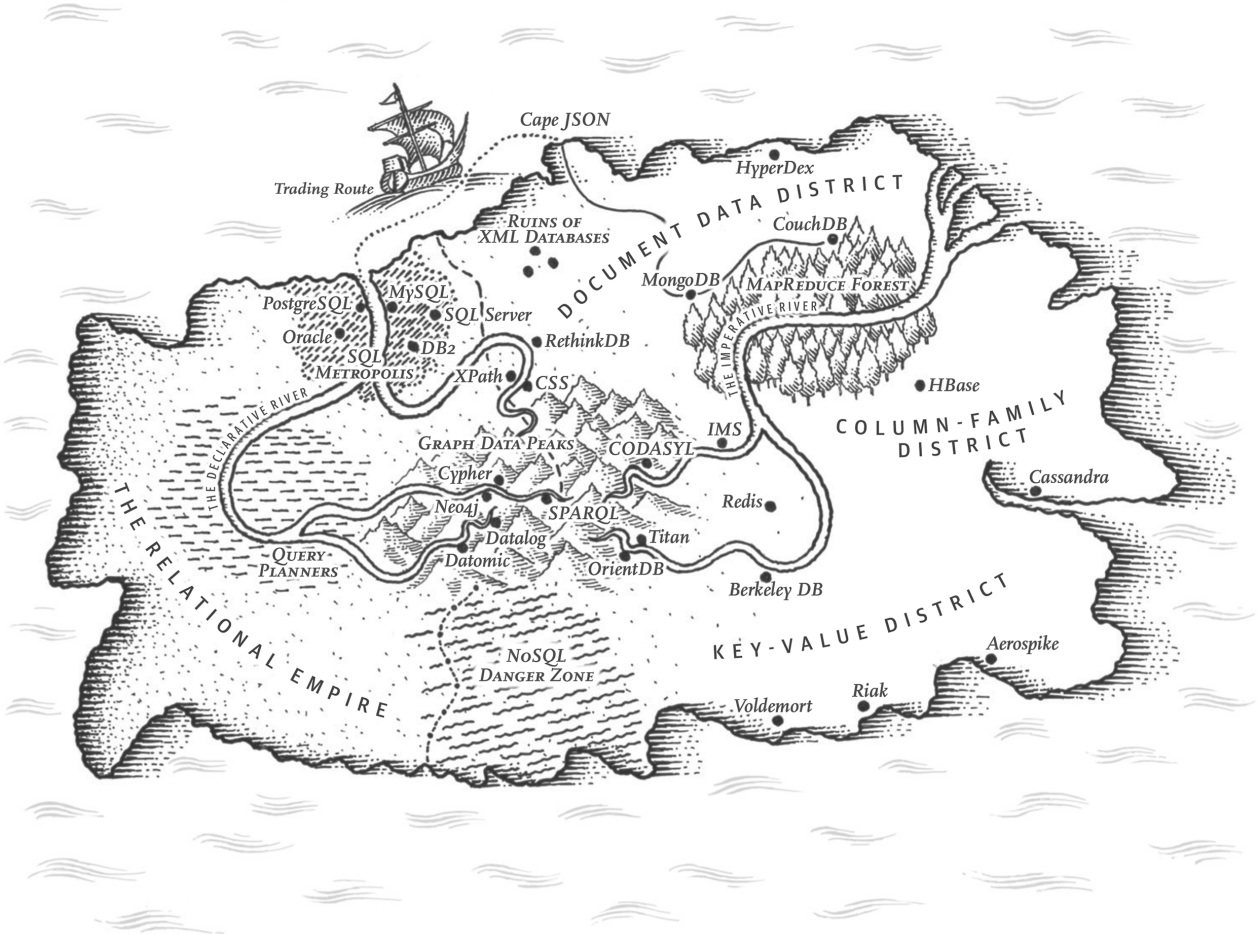


Figure 1: A map of data storage techniques from Designing Data-Intensive Applications [1].

1 This Week

This week our goal is to:

- explore the various techniques developers use to store data;
- investigate the storage options implementing these techniques on the AWS platform;
- run a small application using docker that requires a database; and
- deploy the application in AWS using Terraform.

2 Databases and Data Models

Unfortunately, to build interesting software we often need to store and use data. The storage of data introduces a number of challenges when designing, creating, and maintaining our software. However, not all data storage techniques are created equal; the choice of data storage model can have a profound impact on our software's complexity and maintainability. In this practical, we want to take a superficial exploration of our island of data storage models. For a more in-depth treatment of data storage models that is outside the scope of this course, see Chapter 2 of the *Designing Data-Intensive Applications* book [1].

2.1 Relational Storage

Relational databases what have been exposed to the most in your University career — think MySQL, Postgres, Oracle DB, etc. This type of database is good at modelling the real world which is often a highly connected environment.

Some popular offerings are below:

- MySQL/MariaDB [Amazon RDS / Amazon Aurora].
- Postgres [Amazon RDS / Amazon Aurora].

The AWS offerings of these services come in two different types, we have the traditional approach of server capacity (x cores, y ram) and we have a server-less approach. The server-less approach is a more dynamic database that can scale to large amounts of load when needed though at a cost per request.

2.1.1 ORM

Object Relational Mapping (ORM) is a fairly common tool for programmers to use to make developing with databases smoother. One fairly prevalent example of this is SQLAlchemy which is a very widely used database abstraction for python. SQLAlchemy allows us to move to a higher level of abstraction than SQL queries and perform database actions using standard python code.

The benefits of ORMs are the ability to model database objects in our existing programming language instead of having large blocks of SQL text within our source code. The disadvantages come in when we need to do specific SQL work or where the abstractions cost is greater than the benefits.

2.2 Wide-Column Storage

Wide-Column databases are a form of NoSQL or non-relational data stores. In these data stores the data model design is focused more on having efficient queries at the cost of data duplication. A warning to the reader that these models are not flexible after creation, it is much easier to answer a new use case in a relational model.

- Apache Cassandra [Amazon Keyspaces for Cassandra].
- Apache HBase.

2.3 Key-Value Storage

Key-Value stores are very popular for cache or remote config use cases, some of the most notable are Redis and Memcached. These stores allow efficient lookup of values via keys and are usually stored in-memory.

- Redis [Amazon ElastiCache for Redis].

- Memcached [Amazon ElastiCache for Memcached].
- Amazon DynamoDB.
- Amazon MemoryDB for Redis.

2.4 Time Series Storage

Time series databases are highly focused storage which is tailored to retrieving results by timestamp ranges. Many implementations also take advantage of the data model to allow efficient rollover of data and partitioning. One of the most popular time series databases is Prometheus which is used to store monitoring metrics.

- Amazon Timestream.
- TimescaleDB (Postgres + Addon).
- Prometheus.
- InfluxDB.

2.5 Document Storage

Document databases are a subset of NoSQL databases with a focus on a flexible data model. MongoDB for instance allows the user to store JSON documents and perform queries on those documents. One advantage of document databases is that they match a programmers existing mental model of storing data in formats such as JSON.

- MongoDB.
- Apache CouchDB.
- Amazon DocumentDB.
- Amazon DynamoDB.

2.6 Graph Storage

Graph Databases are relational storage with a few enhancements to allow fast neighbour look-ups. These databases also allow the implementation of graph algorithms to query data.

- Amazon Neptune.
- Neo4J.
- Janus Graph.

3 Deploying an Application with Storage

Thus far in the course we have introduced docker in small packages which do not communicate (much) with the outside world. Today we will be using it to run an application which needs to talk to a database.

1. First we will deploy the application and database locally.
2. Then we will deploy the database on AWS infrastructure and configure our application to talk to the remote database.
3. Finally, we will deploy the application and database on AWS infrastructure.

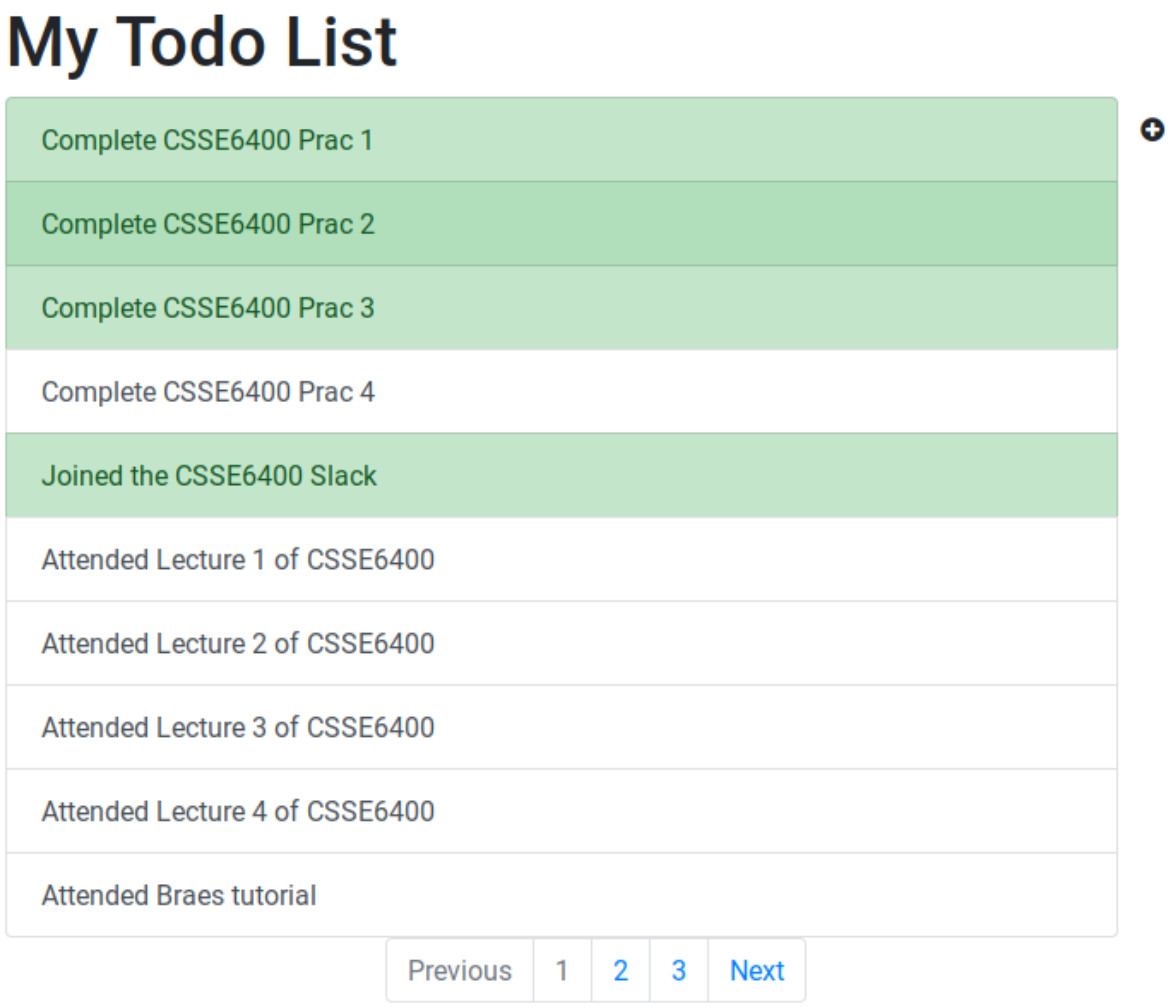


Figure 2: Sample Todo App made for the course

Info

You will need to have docker and docker-compose installed for this practical. Installation will depend on your operating system.

- Docker compose: https://docs.docker.com/compose/install/
- Docker engine: https://docs.docker.com/get-docker/

We also recommend installing the vscode docker plugin or the equlivant tools in IntelliJ IDEs.

Aside

We have only seen individual Docker containers running thus far. As we start to build distributed systems, we will want to use multiple Docker containers which talk to each other. Docker compose is a container orchestration tool that allows us to configure multiple containers to run and setup paths for them to talk to each other.

Warning

For terminal examples in this section, lines that begin with a $ indicate a line which you should type while the other lines are example output that you should expect. Not all of the output is captured in the examples to save on space.

4 Deploying Locally

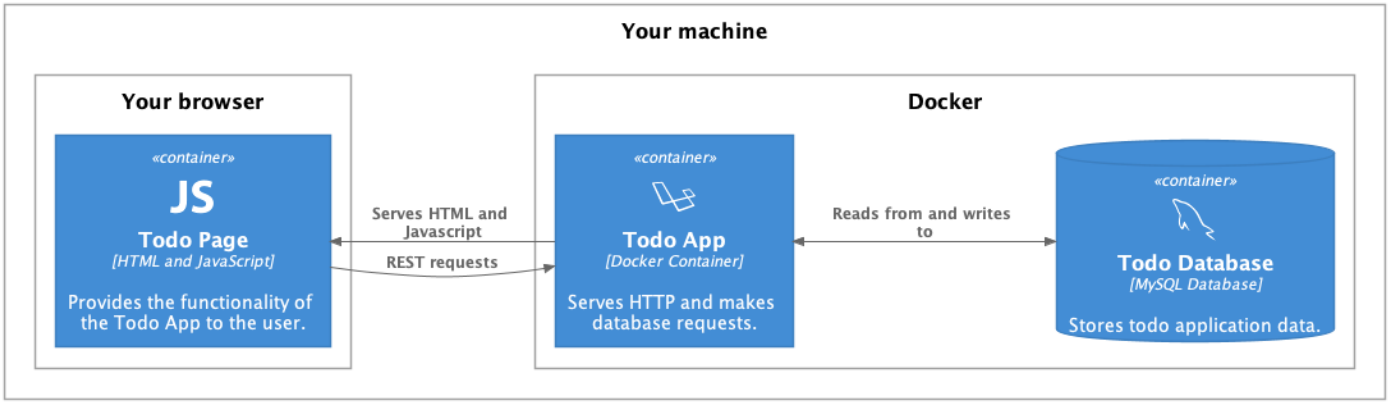


Figure 3: Local deployment diagram

We will be using a container that is pre-built from the Dockerfile below. You should browse the contents of this container to gain more exposure to Dockerfile although you aren't required to use the file for this task — we will use an image already built from this Dockerfile.

```
» cat Dockerfile
FROM ubuntu:21.10
RUN apt-get update \
```

```
       && DEBIAN_FRONTEND=noninteractive apt install -y \
           php \
           php-mysql \
           php-xml \
           php-curl \
           curl \
           git \
           unzip
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin
   --filename=composer
COPY . /app
WORKDIR /app
RUN composer install
CMD ["php", "artisan", "serve", "--host=0.0.0.0"]
```

Our first task is to have a local instance of the Todo App including the database. To get started we need to make a new directory for our work and create a Docker compose file.

```
$ mkdir prac4 && cd prac4
$ touch docker-compose.yml
```

Docker Compose is a small helper utility that allows us to more easily run docker applications without needing to remember a lot of command line parameters. Instead we define how we want our docker container to run through a YAML config file. Insert the following into your docker-compose.yml file.

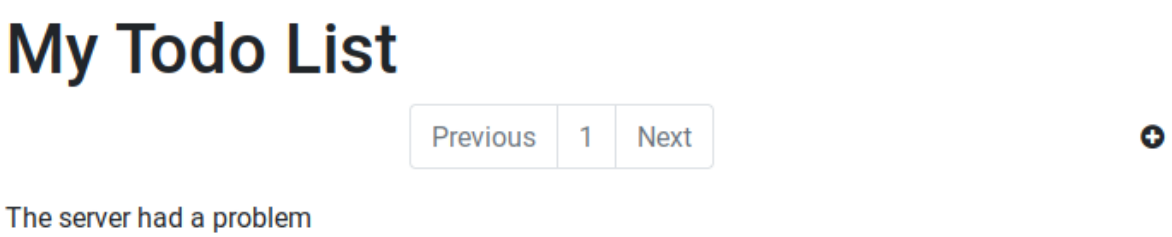
```
   » cat docker-compose.yml
version: '3.3'
services:
  backend:
    image: ghcr.io/csse6400/todo-app:latest
    ports:
      - '8000:8000'
    environment:
      APP_ENV: 'local'
      APP_KEY: 'base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A='
      APP_DEBUG: 'true'
      LOG_LEVEL: 'debug'
```

A few things to point out in the file. We have defined a single service (container) called backend which uses the pre-built docker image from `ghcr.io/csse6400/todo-app` with the tag of latest. We then have exposed this onto our machine on port 8000 and have passed a few environment variables required for the application.

`docker-compose up` will use our docker compose configuration to run any containers we have specified.

```
$ docker-compose up
Creating network "p1_default" with the default driver
Creating p1_backend_1 ... done
Attaching to p1_backend_1
backend_1 | Starting Laravel development server: http://0.0.0.0:8000
backend_1 | [Sun Mar 20 07:56:23 2022] PHP 8.0.8 Development Server (http
    ://0.0.0.0:8000) started
```

Now we head to our browser and go to http://127.0.0.1:8000, you should be presented with the following screen.



When we see unhelpful error messages such as this, we should investigate the web console in our browser. The web console will tell us which network calls are being made and which calls are causing the error message. Follow along with your tutor to find that the offending network call is to http://127.0.0.1:8000/api/v1/todo.

Once we open that address we have a clearer idea of what has gone wrong. The page you should see is shown in Figure 4 and the reason is that our backend cannot connect to a database (because we have yet to make one).

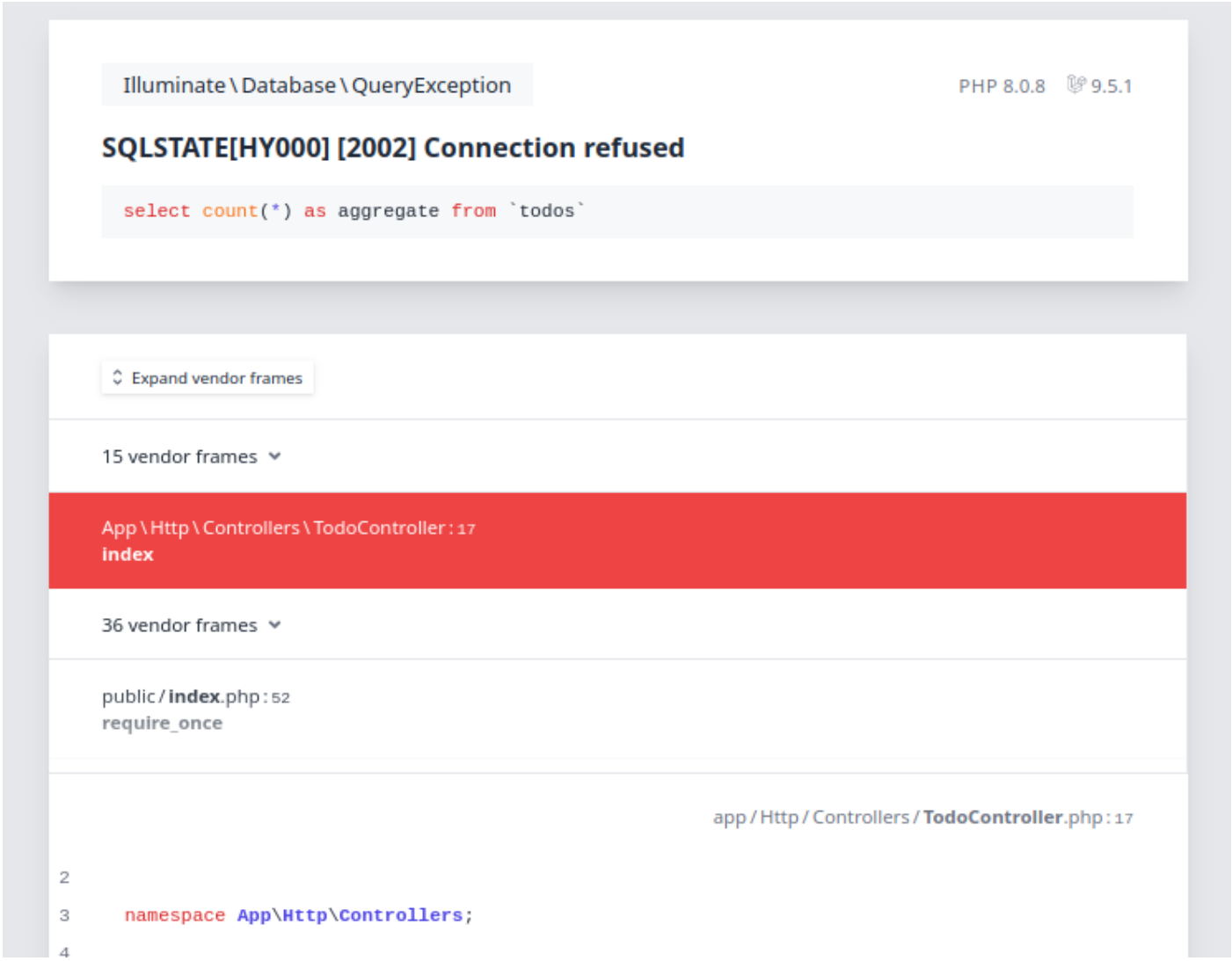


Figure 4: The expected error page when accessing `http://127.0.0.1:8000/api/v1/todo`.

To fix this lets add a popular relation database, MySQL, to our docker-compose file. Edit your docker compose file to match as shown below.

```
» cat main.tf
version: '3.3'
services:
  db:
    image: mysql:8-debian
    environment:
      MYSQL_DATABASE: 'todoapp'
      MYSQL_USER: 'todoapp'
      MYSQL_PASSWORD: 'password'
      MYSQL_ROOT_PASSWORD: 'password'
    ports:
      - '3306:3306'

  backend:
```

```
    image: ghcr.io/csse6400/todo-app:latest
    depends_on:
      - db
    ports:
      - '8000:8000'
    environment:
      APP_ENV: 'local'
      APP_KEY: 'base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A='
      APP_DEBUG: 'true'
      LOG_LEVEL: 'debug'
      DB_CONNECTION: 'mysql'
      DB_HOST: 'db'
      DB_PORT: '3306'
      DB_DATABASE: 'todoapp'
      DB_USERNAME: 'todoapp'
      DB_PASSWORD: 'password'
```

Now we have two services (containers) for our app and we have added a few more environment variables for our backend to know how to connect to the database. The `DB_HOST` variable uses a feature of docker compose where you can refer to other services by their name rather than a traditional IP address. This makes it easy for us to setup communication between these two services.

From the same shell let's re-run our containers, you may need to CTRL+C to stop the current running containers. Once they have shutdown, run the up command again.

```
$ docker-compose up
Starting p2_db_1 ... done
Starting p2_backend_1 ... done
Attaching to p2_db_1, p2_backend_1
db_1 | 2022-03-20 08:11:55+00:00 [Note] [Entrypoint]: Entrypoint ....
db_1 | 2022-03-20 08:11:55+00:00 [Note] [Entrypoint]: Switching t....
db_1 | 2022-03-20 08:11:55+00:00 [Note] [Entrypoint]: Entrypoint ....
db_1 | 2022-03-20T08:11:55.438996Z 0 [System] [MY-010116] [Server....
db_1 | 2022-03-20T08:11:55.445261Z 1 [System] [MY-013576] [InnoDB....
backend_1 | Starting Laravel development server: http://0.0.0.0:8000
db_1 | 2022-03-20T08:11:55.535803Z 1 [System] [MY-013577] [InnoDB....
db_1 | 2022-03-20T08:11:55.673757Z 0 [Warning] [MY-010068] [Serve....
db_1 | 2022-03-20T08:11:55.673784Z 0 [System] [MY-013602] [Server....
db_1 | 2022-03-20T08:11:55.674810Z 0 [Warning] [MY-011810] [Serve....
db_1 | 2022-03-20T08:11:55.684729Z 0 [System] [MY-010931] [Server....
db_1 | 2022-03-20T08:11:55.684756Z 0 [System] [MY-011323] [Server....
backend_1 | [Sun Mar 20 08:11:55 2022] PHP 8.0.8 Development Serv....
```

Now when we go to http://127.0.0.1:8000/api/v1/todo we see a different error message, as shown in Figure 5. This error is complaining that we have a database that we can connect to but the todos table doesn't exist.

To populate the database our application comes with database migration files. One way would be to click the "RUN MIGRATIONS" button shown on the error page but we also want to pre-populate our database with some dummy data as well.

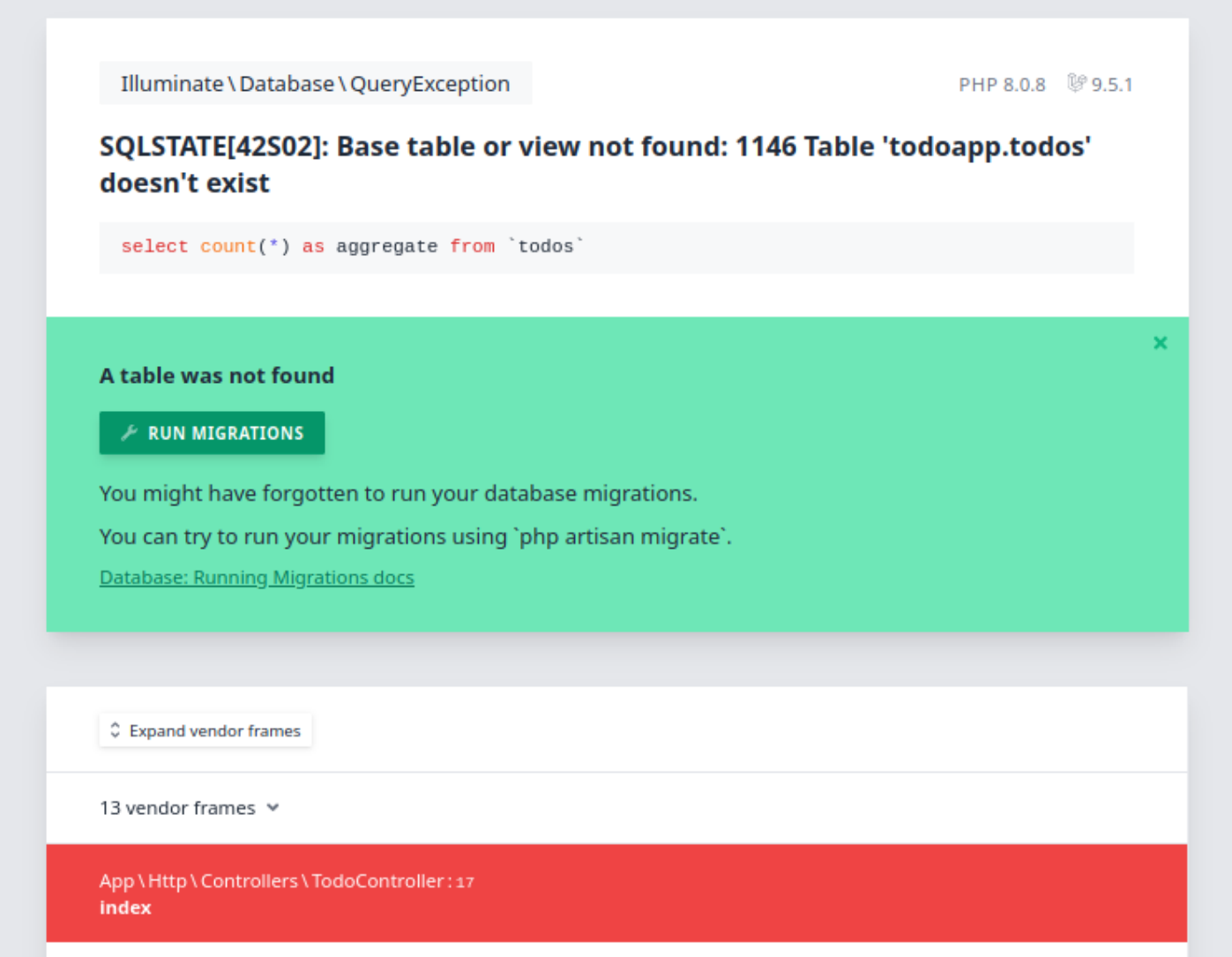
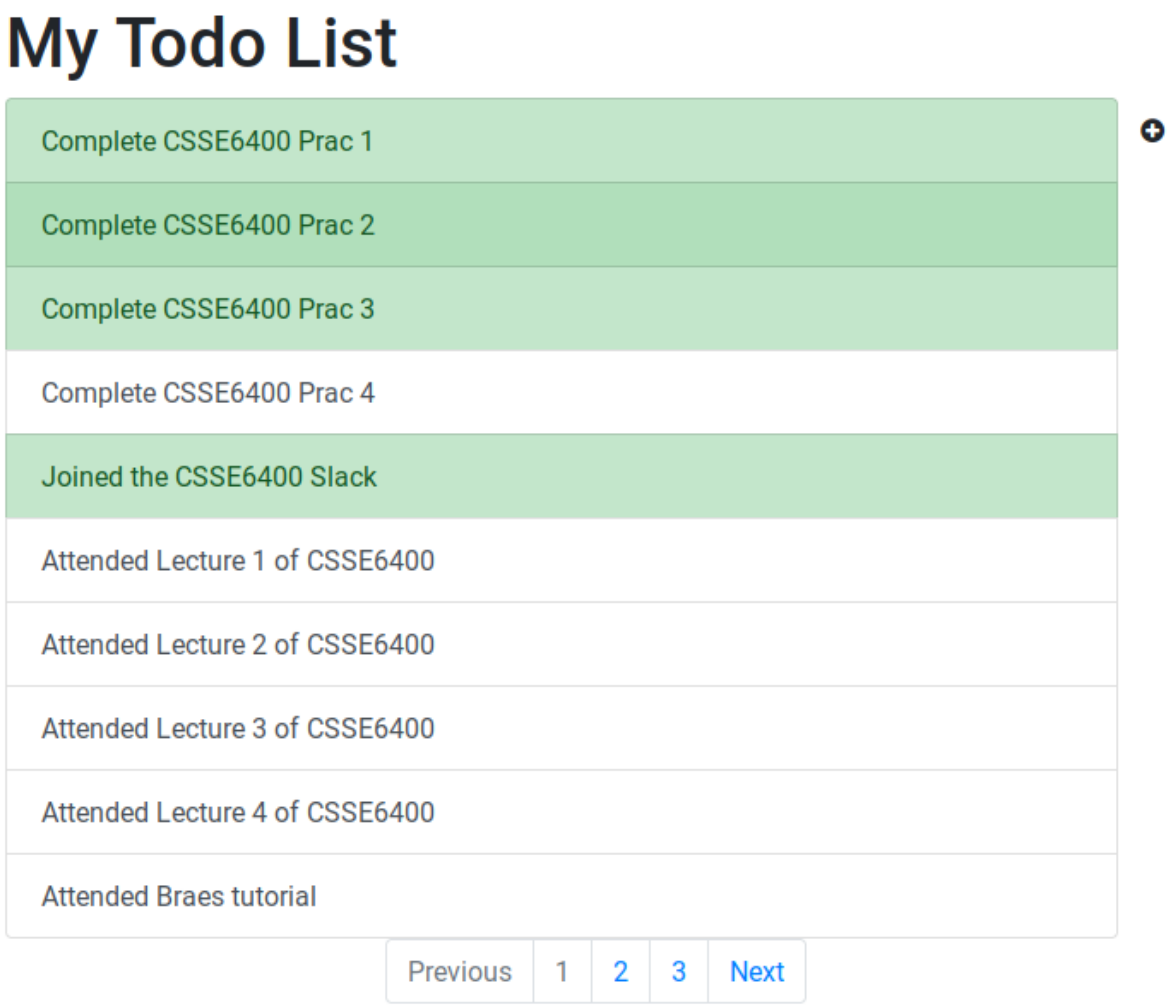


Figure 5: The expected error page when accessing `http://127.0.0.1:8000/api/v1/todo` after creating a database.

To do this we are going to jump into the running container and execute the migrations ourselves, using `docker-compose exec`. Start by opening a new terminal so that we can leave the docker containers running. In this new terminal navigate to **the directory where** `docker-compose.yml` **lives** and run the following command:

```
$ docker-compose exec backend php artisan migrate:fresh --seed
Dropped all tables successfully.
Migration table created successfully.
Migrating: 2022_03_19_041557_create_todos_table
Migrated: 2022_03_19_041557_create_todos_table (7.55ms)
Seeding: Database\Seeders\TodoSeeder
Seeded: Database\Seeders\TodoSeeder (6.56ms)
Database seeding completed successfully.
```

Now with this run we can check back at our web app (at `http:127.0.0.1:8000`)and you should see a fully functional todo app.



Info

The database migrations are performed by Laravel which is a popular PHP web framework. In the database it has created a table to keep track of which migrations have already been run so that it will skip them latter. To enable this functionality you will have to edit migrate:fresh to just migrate.

For a production instance you would also typically remove the –seed parameter as you do not want to insert dummy data into your production database.

4.1 Exercise: Migrations at startup

So far when we run this application we have to perform the database migrations manually. To help us get up and running we are going to make a small modification to pre-run the migrations when are web app starts. First we need to have a look at how the container is set to launch by default. In the Dockerfile attached at the start of the practical we see that we have defined the command to run on the last line with the CMD directive.

```
» cat Dockerfile
```

```
FROM ubuntu:21.10
...
...
...
CMD ["php", "artisan", "serve", "--host=0.0.0.0"]
```

Info

When working with docker it can get confusing around the networking aspects. In this application I have specified that the server must listen on all network interfaces (0.0.0.0). Without this flag the default is 127.0.0.1 which even though its the localhost the forwarded traffic through the docker container would never reach it.

This command launches the laravel development server and listens on all interfaces on the host. We are going to override this in our docker-compose file so that we run the migrations then start the server. Add the following line to the docker-compose.yml that you have been developing during the practical.

```
command: sh -c "sleep 30 && php artisan migrate:refresh --seed && php artisan serve
    --host=0.0.0.0"
```

This new command does the following:

- Waits for the database to be ready in a simple way.
- Runs the migrations and seeds the database, as we have seen earlier.
- Starts the development server as the container originally did.

Example: condensed version of the goal `docker-compose.yml` attached below.

```
» cat docker-compose.yml
version: '3.3'
services:
  db:
    ...

  backend:
    ...
    environment:
      ...
    command: sh -c "sleep 10 && php artisan migrate:refresh --seed && php artisan
       serve --host=0.0.0.0"
```

Now when we launch the docker-compose we can see that our migrations were run in the output.

```
$ docker-compose up
...
...
backend_1 | Rolling back: 2022_03_19_041557_create_todos_table
backend_1 | Rolled back: 2022_03_19_041557_create_todos_table (8.28ms)
backend_1 | Migrating: 2022_03_19_041557_create_todos_table
backend_1 | Migrated: 2022_03_19_041557_create_todos_table (11.55ms)
backend_1 | Seeding: Database\Seeders\TodoSeeder
backend_1 | Seeded: Database\Seeders\TodoSeeder (44.77ms)
backend_1 | Database seeding completed successfully.
backend_1 | Starting Laravel development server: http://0.0.0.0:8000
backend_1 | [Sun Mar 20 12:08:41 2022] PHP 8.0.8 Development Server (http
    ://0.0.0.0:8000) started
```

We can also bake this into the container by extending the original, it is fairly common to see projects in the wild that run a init script when the container launches. An exercise left for the reader is to build upon the provided docker container by including an init script.

5 Deploying a Database in AWS

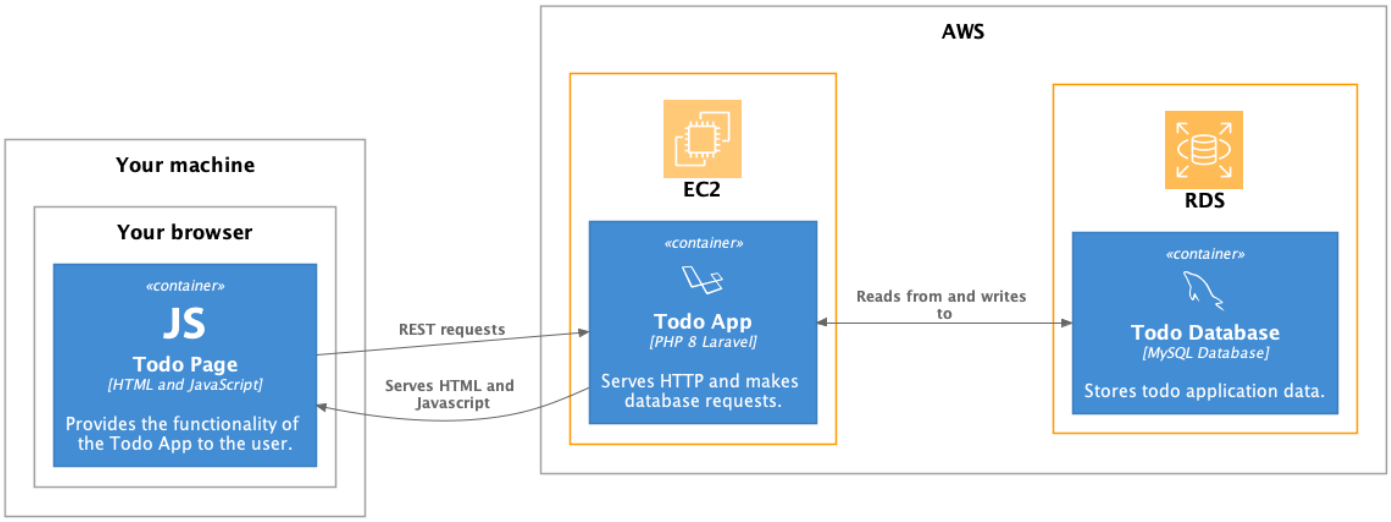
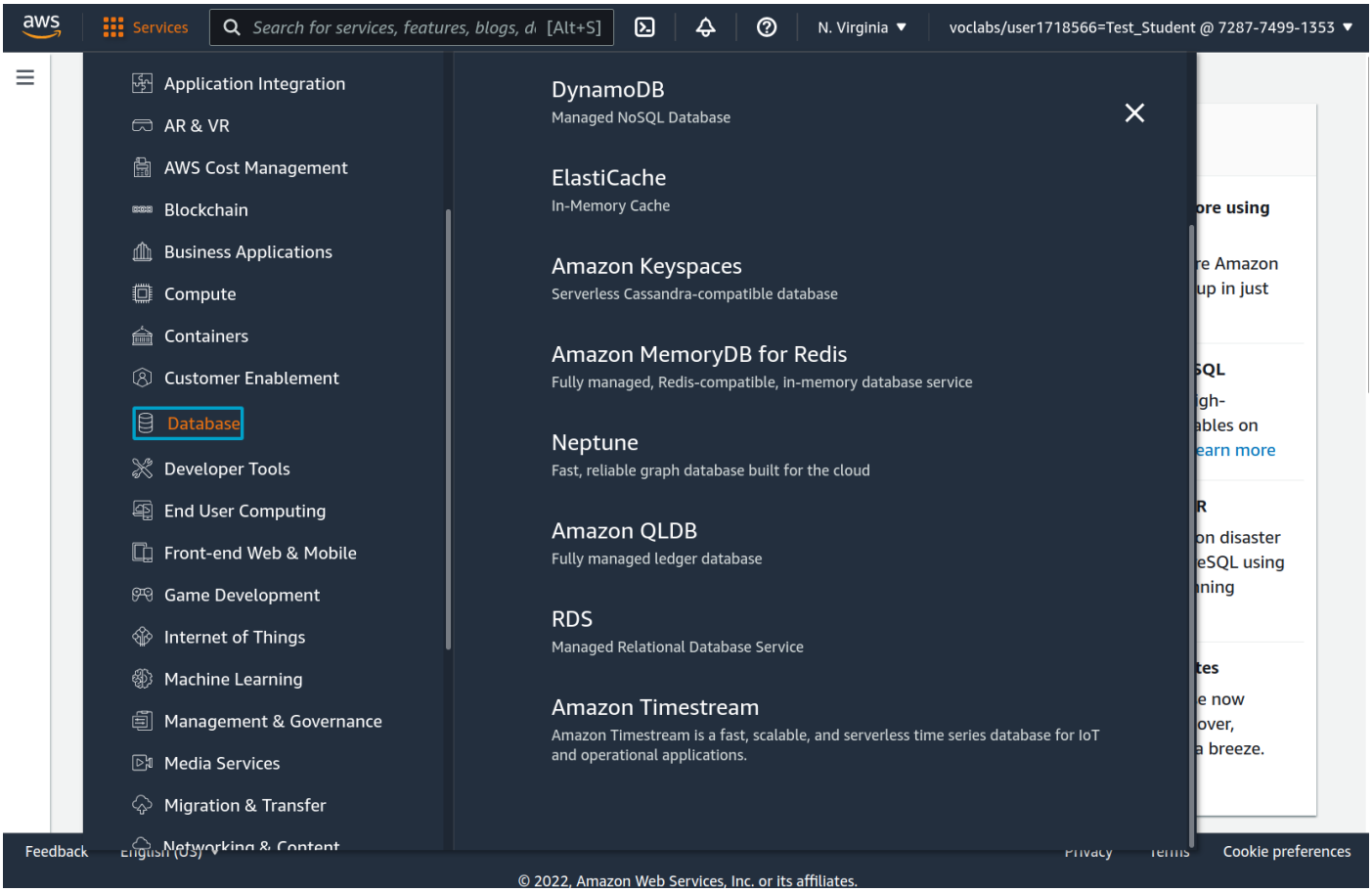


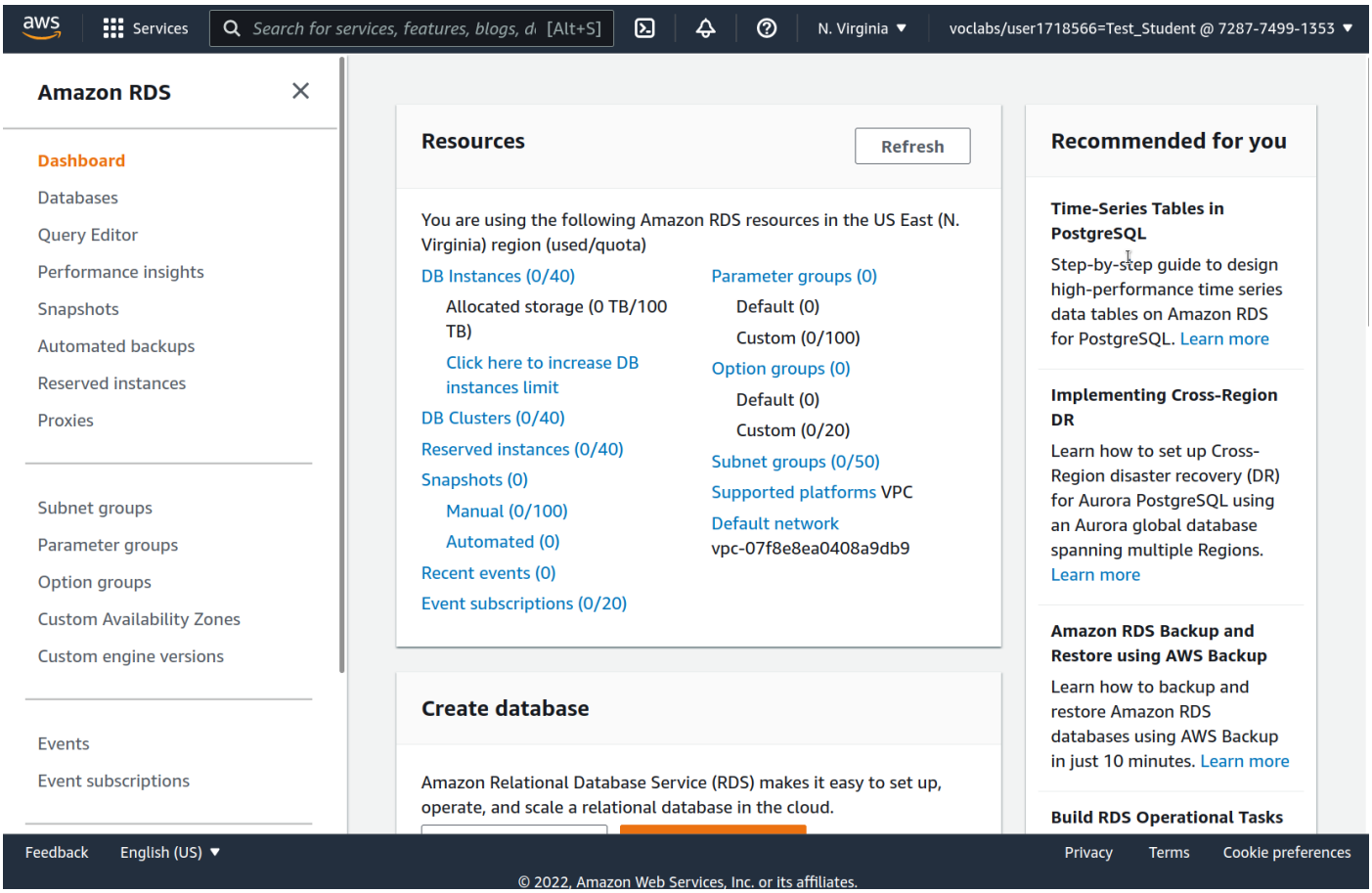
Figure 6: Remote database deployment diagram

Now we have a locally running Todo App let's move to AWS, start up your Learner Lab environment now. 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 to Section 6 for the terraform setup.

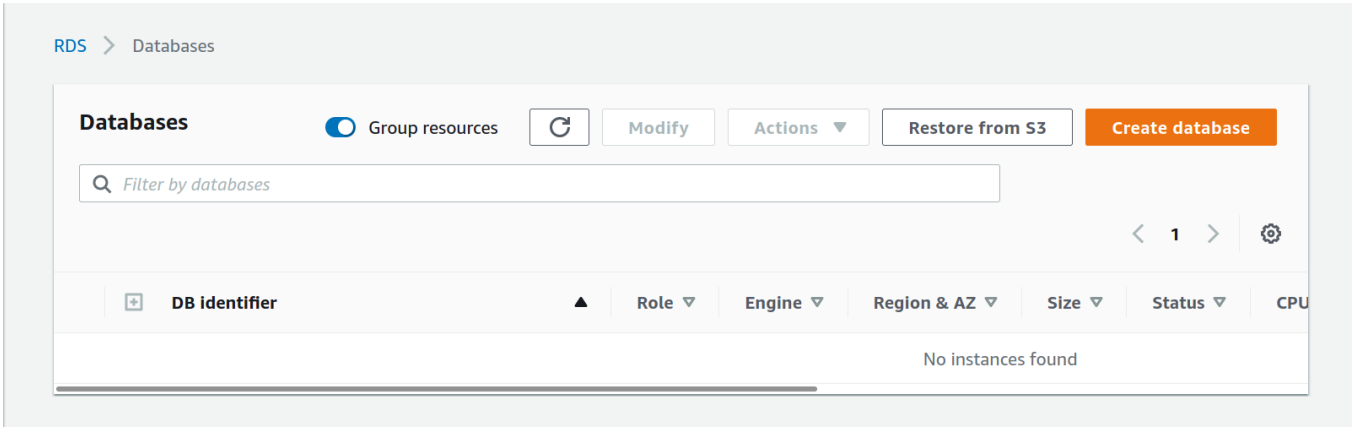
To get started let's jump into the lab environment and have a look at AWS RDS which is an AWS managed database service. To get to the RDS service either search it or browse Services -> Database -> RDS as shown below.



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)”.



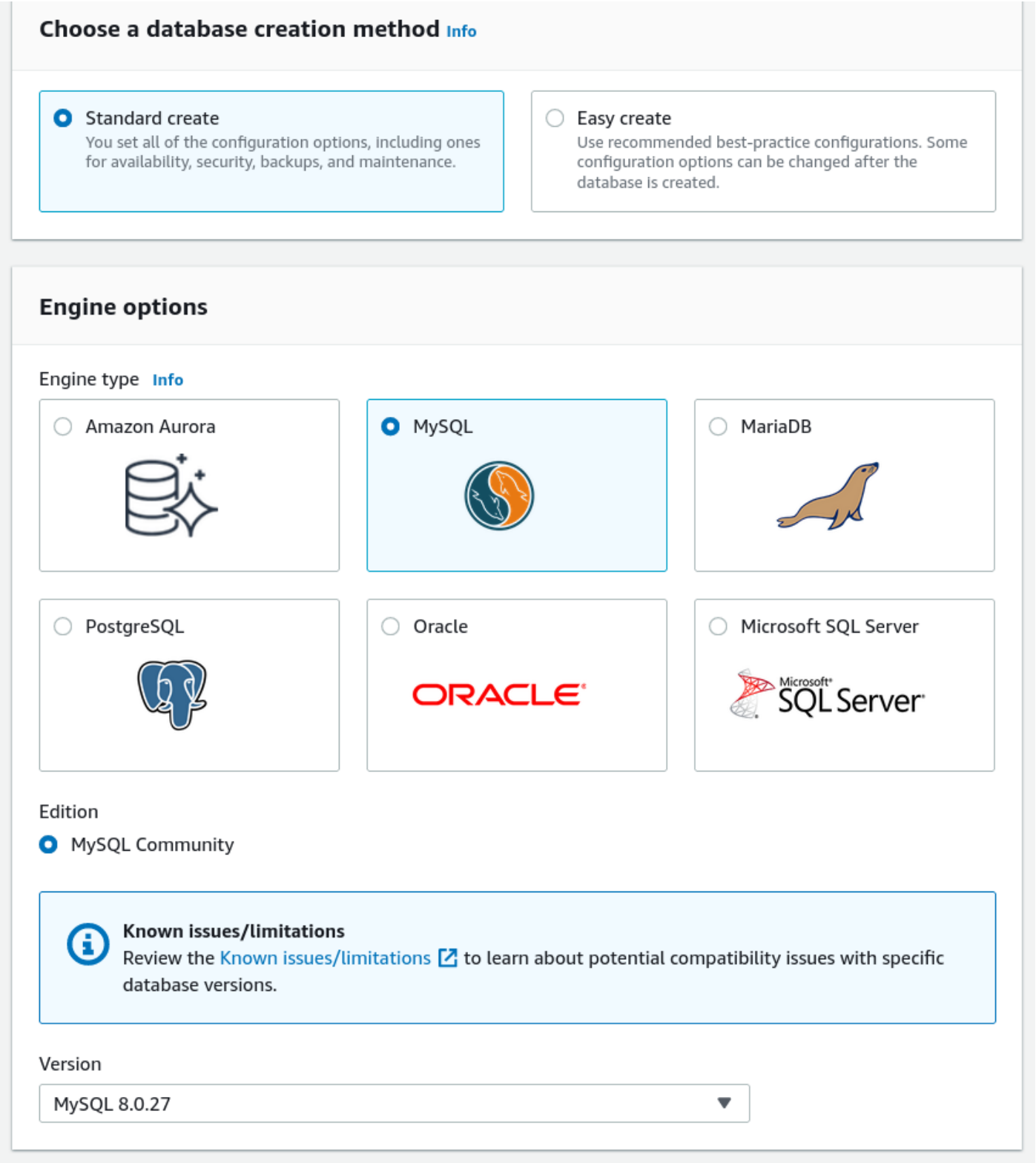
This page should appear familar as it's very similar to the AWS EC2 instance page. Let us create a new database by hitting the "Create Database" button.



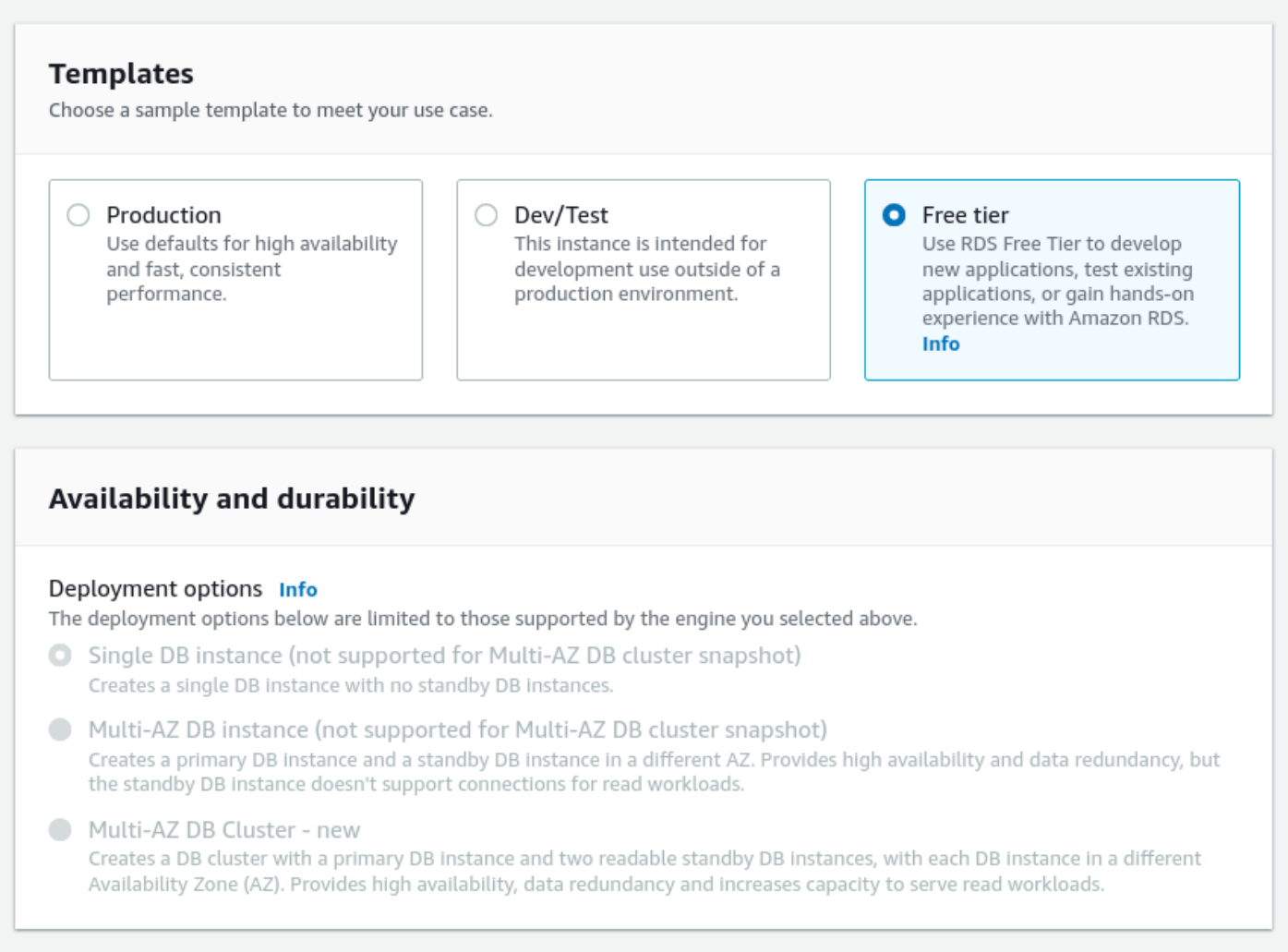
Warning

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.

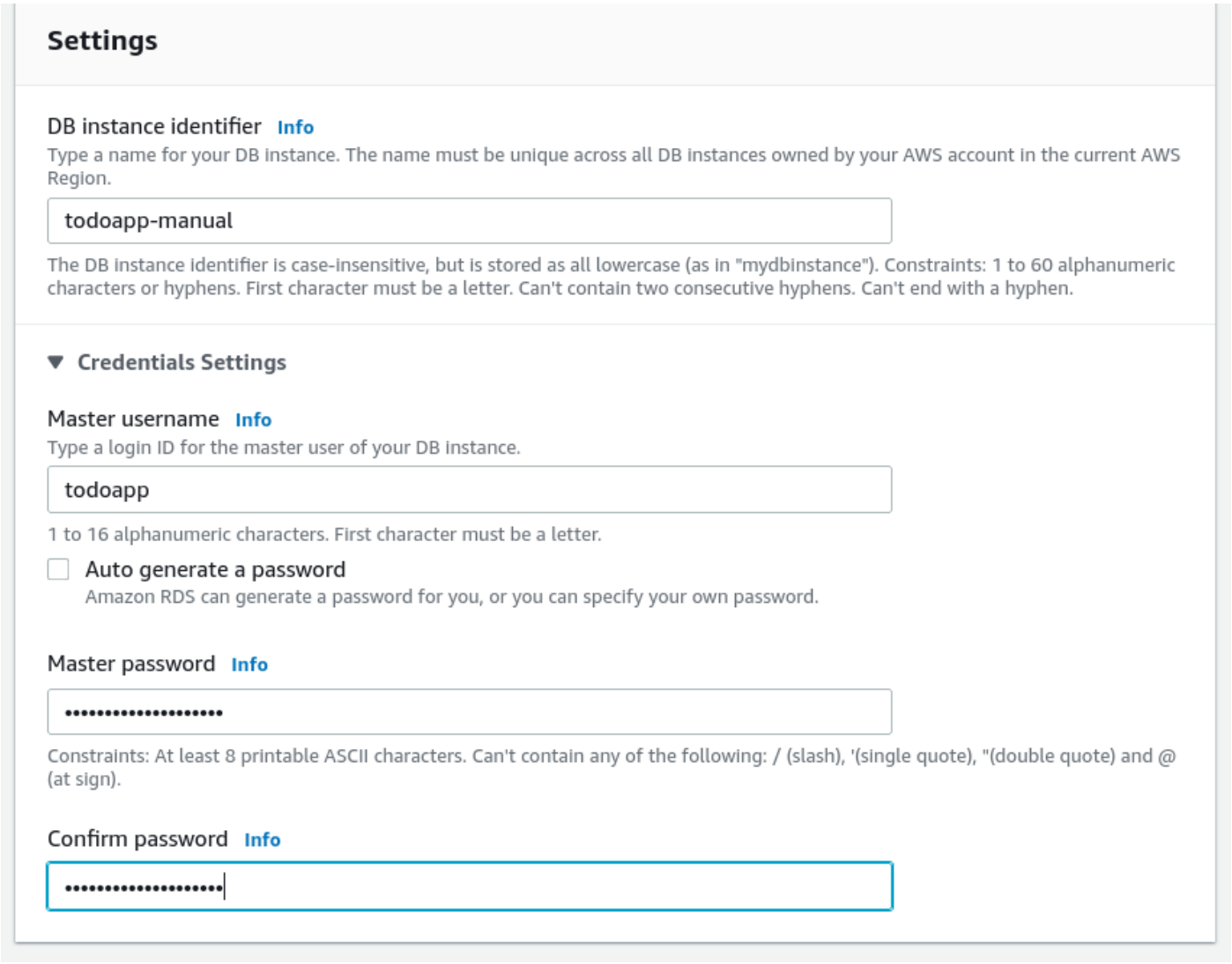
We will be creating a standard database so select standard and MySQL. We will use version 8 to match the local version.



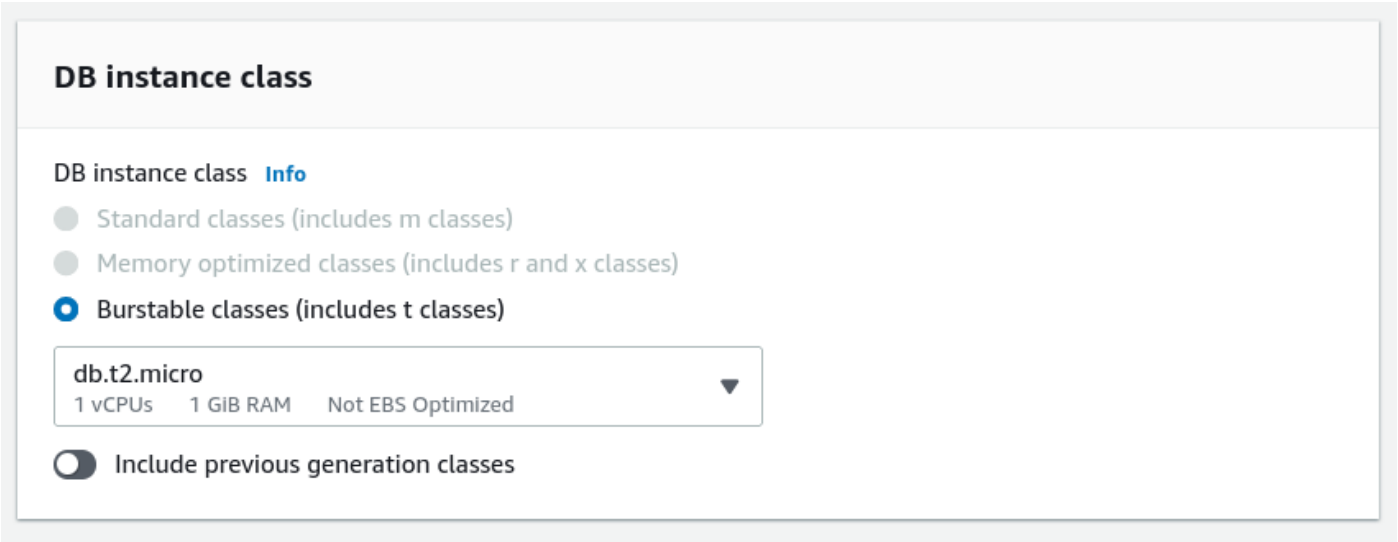
For today we are going to use “Free Tier” but in the future, you may wish to explore the different deployment options. Please peruse the available different options.



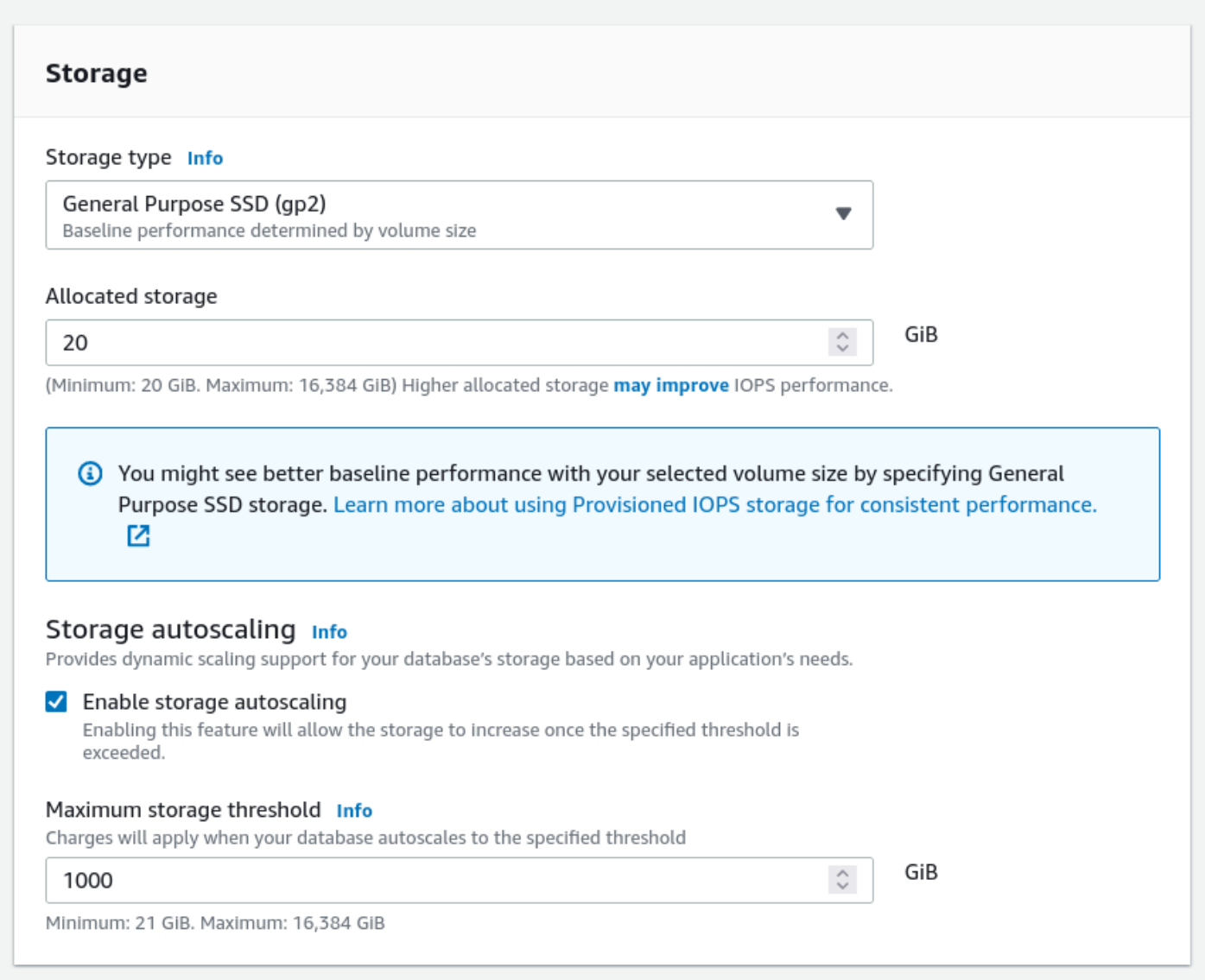
Now we need to name our database and create credentials to connect via. Please enter a reasonable password and keep this aside for later. We will need it for our local docker-compose file.



We will use the default class type, t2.micro, which should be sufficient for this practical.

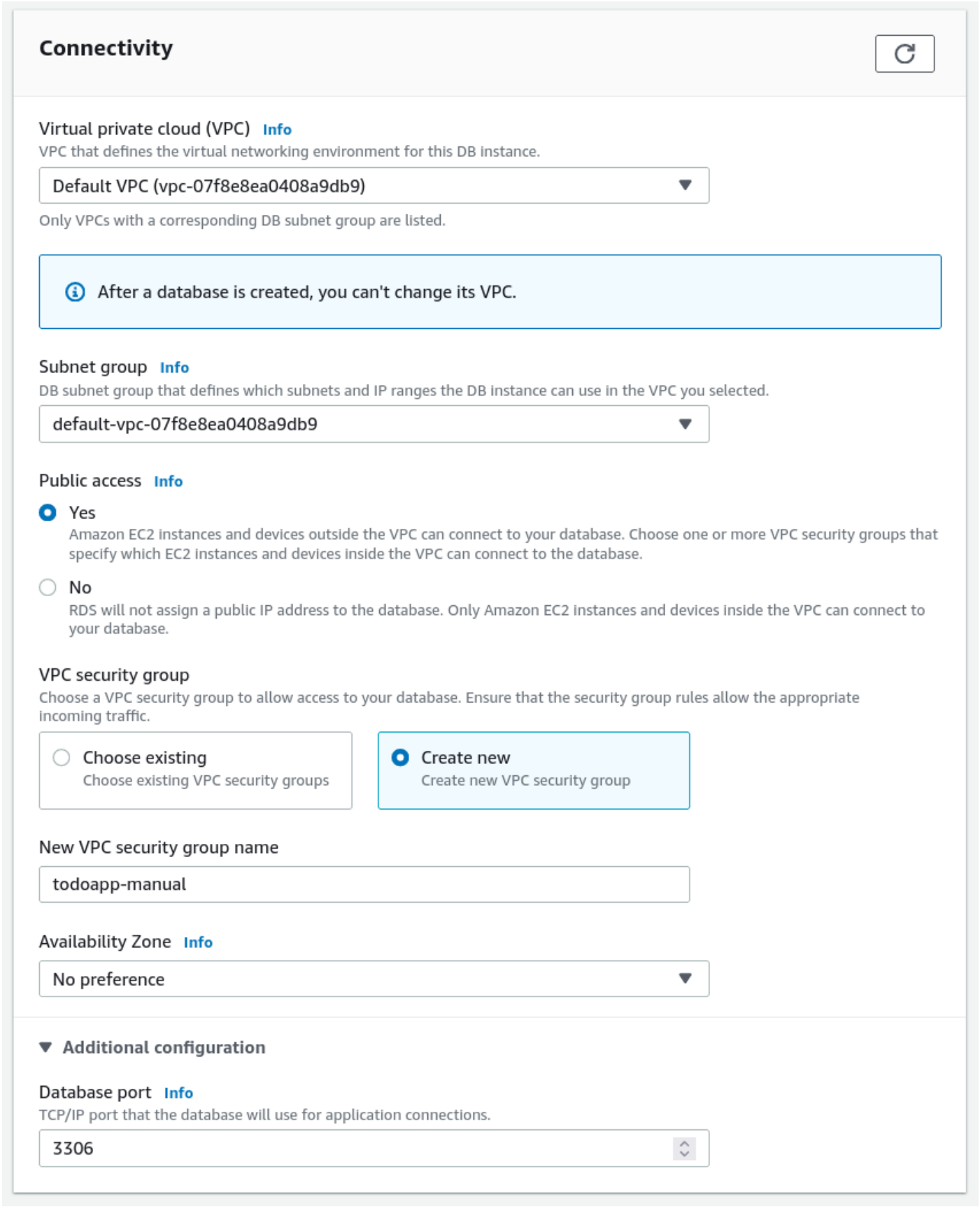


For storage we will leave all the default options.

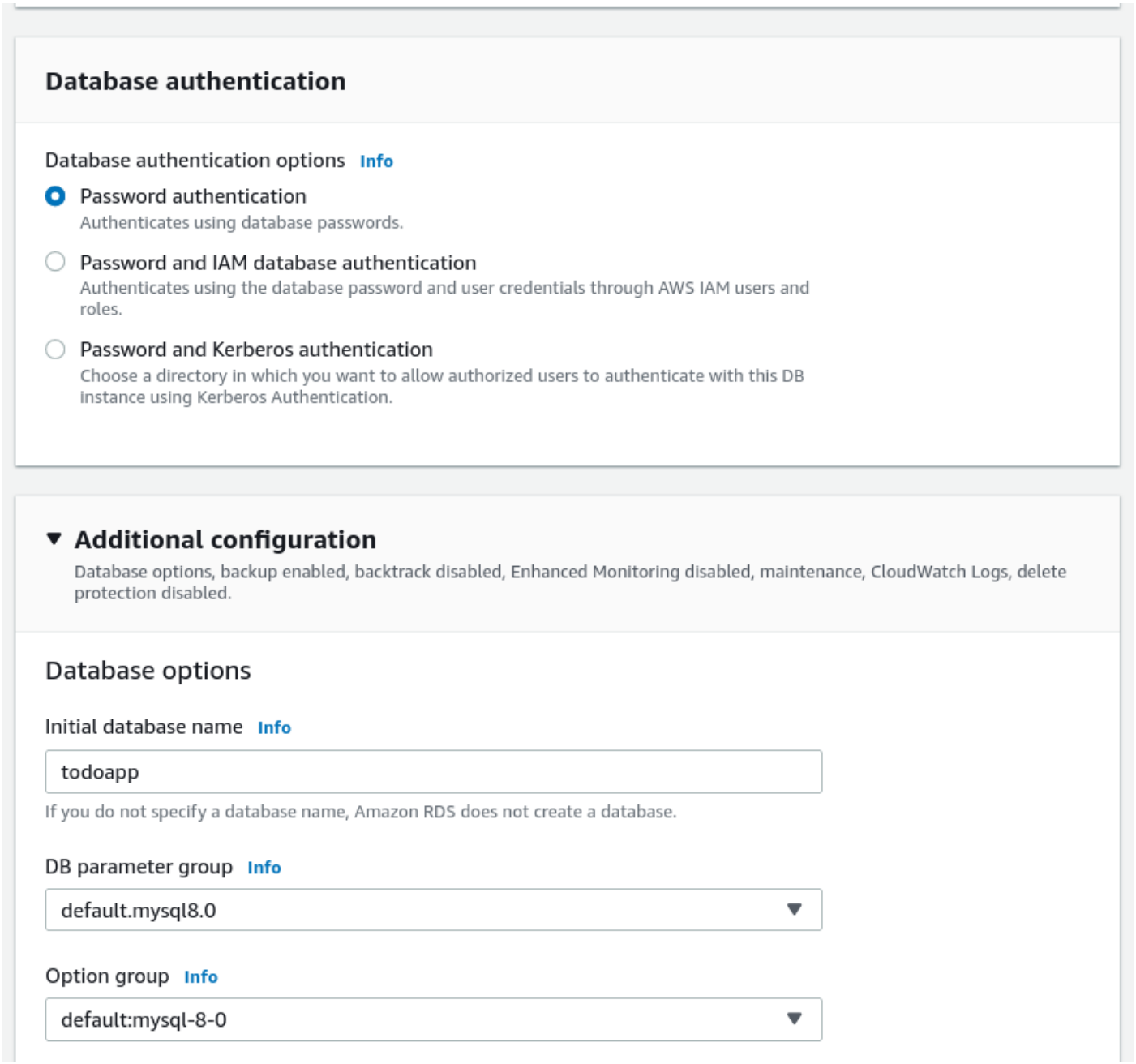


In connectivity we need to make sure our instance is publicly available. Usually you don't want to expose your databases publicly and, would instead, have a web server sitting in-front. But for today we will be running that web server locally so for convenience we need public access.

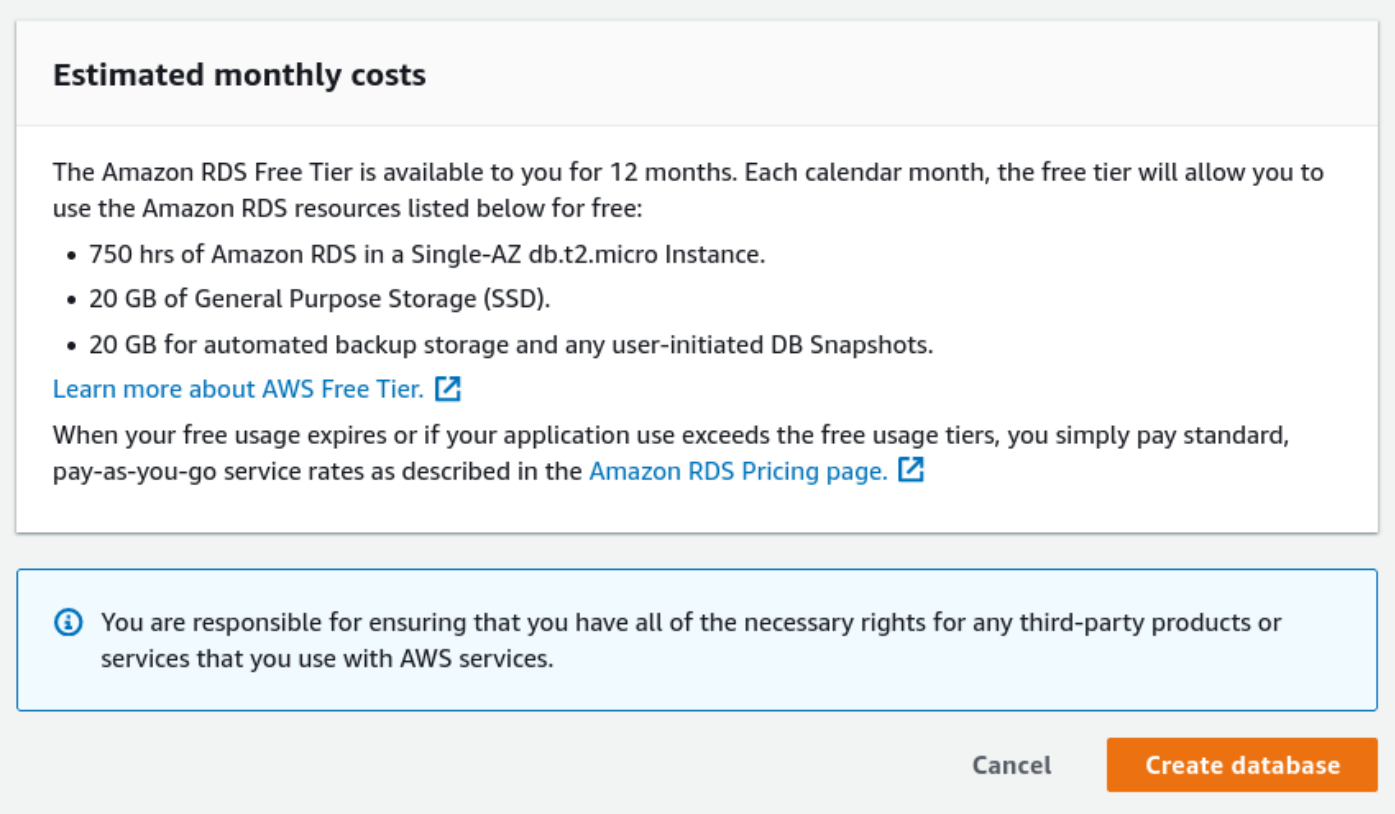
When selecting public access as yes we have to create a new Security Group, give this Security Group a sensible name.



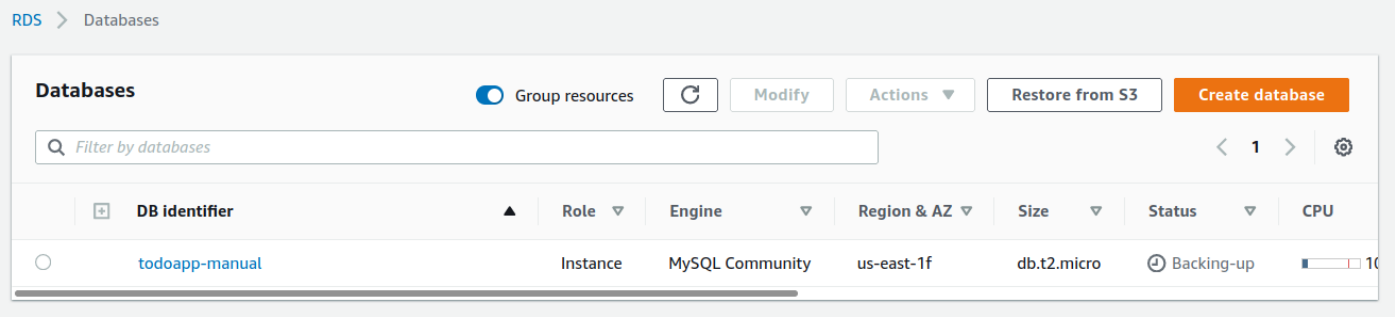
We will leave the authentication as password based but we need to expand the “Additional configuration”. Fill in the “Initial Database Name” section to be “todoapp”, this is similar to what we had in the Docker Compose.



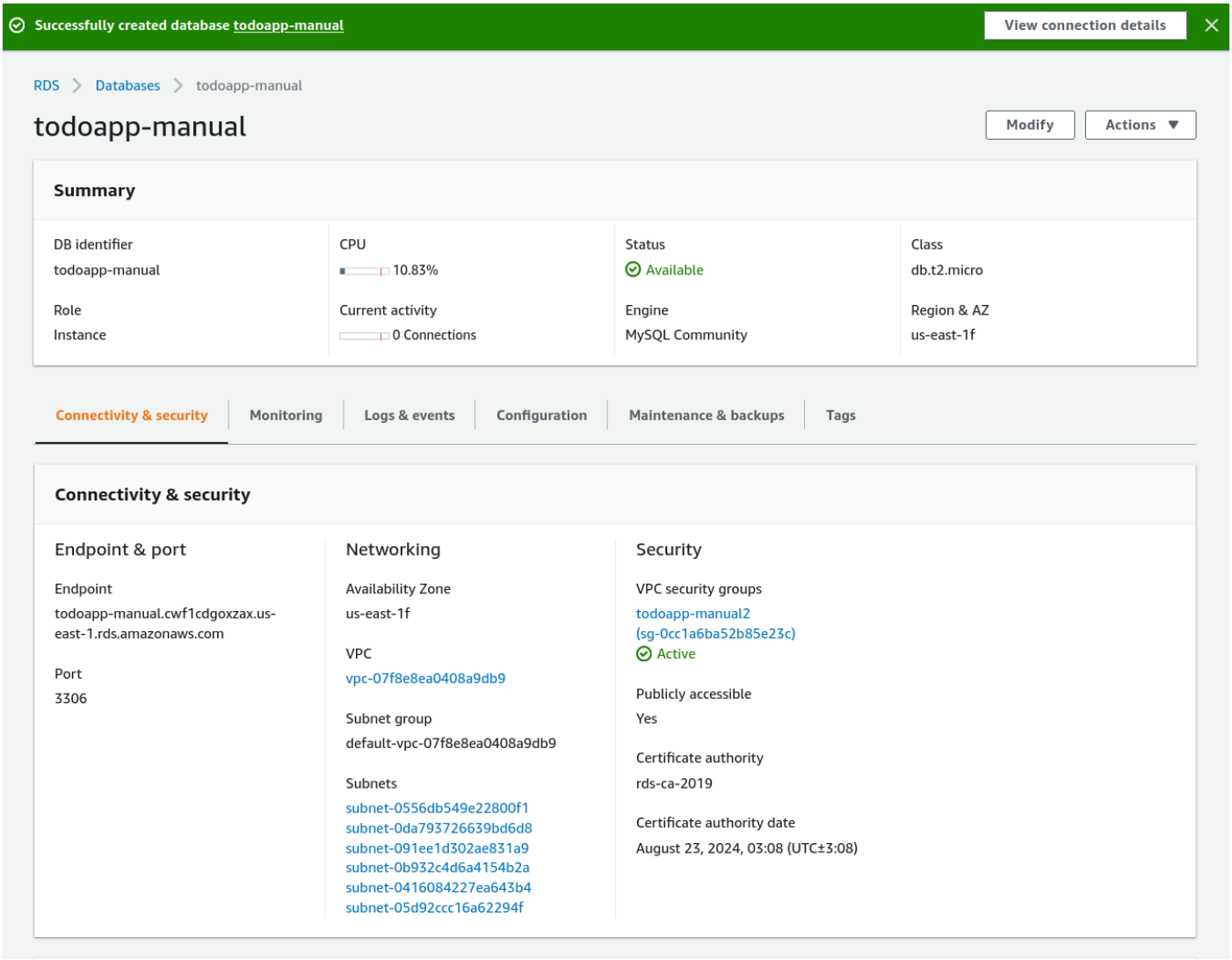
Now we can click create which will take some time.



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.



So with the database all ready to go, let's get started on running our app. First thing to do is to stop any running instances you have and then delete the database service from the docker compose. Now we want to edit the `DB_HOST` and `DB_PASSWORD` to match what we set in AWS. Below is an example of my configuration.

```
» cat docker-compose.yml
version: '3.3'
services:
  backend:
    image: ghcr.io/csse6400/todo-app:latest
    ports:
      - '8000:8000'
    environment:
      APP_ENV: 'local'
      APP_KEY: 'base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A='
      APP_DEBUG: 'true'
      LOG_LEVEL: 'debug'
      DB_CONNECTION: 'mysql'
      DB_HOST: 'todoapp-manual.cwf1cdgoxzax.us-east-1.rds.amazonaws.com'
      DB_PORT: '3306'
```

```
      DB_DATABASE: 'todoapp'
      DB_USERNAME: 'todoapp'
      DB_PASSWORD: 'MyVerySecurePassword'
    command: sh -c "sleep 10 && php artisan migrate:refresh --seed && php artisan
       serve --host=0.0.0.0"
```

Now we can run `$ docker-compose up`. This will take a lot longer than before, the reason behind this is that by default our lab is running in the US and it takes a lot longer to communicate to a DB half way across the world.

```
$ docker-compose up
Creating network "remote_default" with the default driver
Creating remote_backend_1 ... done
Attaching to remote_backend_1
backend_1 | Migration table not found.
backend_1 | Migration table created successfully.
backend_1 | Migrating: 2022_03_19_041557_create_todos_table
backend_1 | Migrated: 2022_03_19_041557_create_todos_table (612.48ms)
backend_1 | Seeding: Database\Seeders\TodoSeeder
backend_1 | Seeded: Database\Seeders\TodoSeeder (13,930.36ms)
backend_1 | Database seeding completed successfully.
backend_1 | Starting Laravel development server: http://0.0.0.0:8000
backend_1 | [Mon Mar 21 04:45:34 2022] PHP 8.0.8 Development Server (http
   ://0.0.0.0:8000) started
```

We can now also browse the app as before but with just a lot more lag.

6 Deploying in AWS

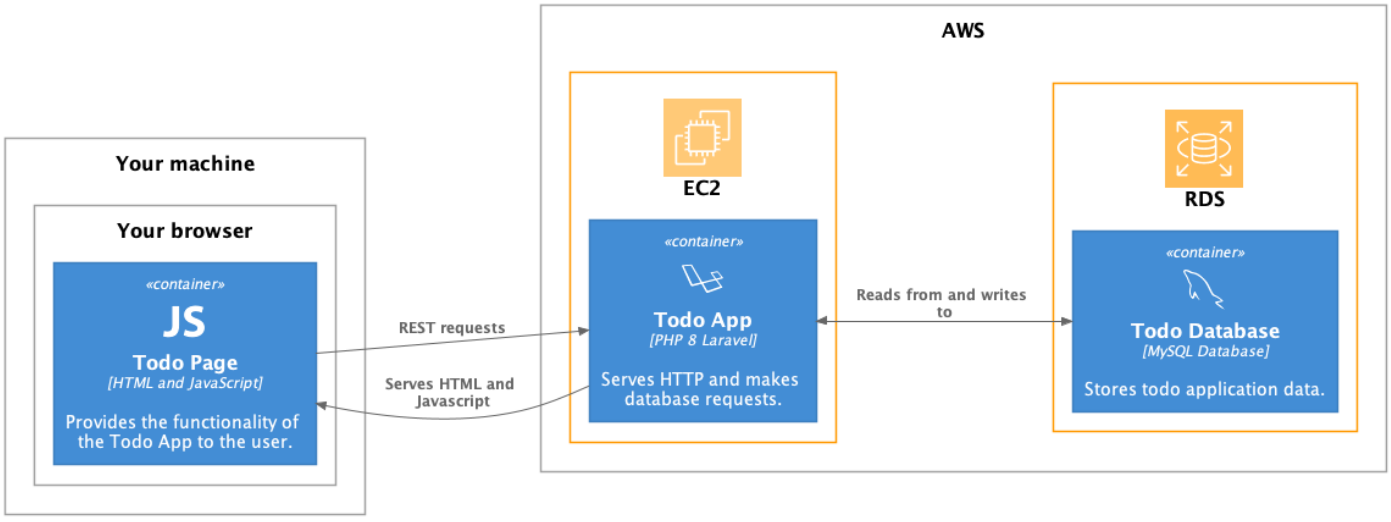


Figure 7: Remote deployment diagram

In this section we will deploy the application and database in AWS. Rather than locally configuring a database as we did in the previous section, we will deploy everything with Terraform.

6.1 Authentication

As we did last week, specify the AWS provider we will use and download the Learner Lab credentials into a `credentials` file. Once you have setup your `main.tf` and `credentials` files, run `terraform init`.

```
» cat main.tf
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_file = "./credentials"
}
```

6.2 RDS Database

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 = "mysql"
  engine_version = "8.0.27"
  instance_class = "db.t2.micro"
  name = "todoapp"
  username = "todoapp"
  password = local.password
  parameter_group_name = "default.mysql8.0"
  skip_final_snapshot = true
  vpc_security_group_ids = [aws_security_group.todoapp-database.id]
  publicly_accessible = true

  tags = {
    Name = "todoapp-database"
```

```
  }
}
```

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

```
» cat main.tf
resource "aws_security_group" "todoapp-database" {
  name = "todoapp-database"
  description = "Allow inbound MySQL traffic"

  ingress {
    from_port = 3306
    to_port = 3306
    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 = "todoapp-database"
  }
}
```

6.3 Container on AWS

As we mentioned in the Infrastructure as Code notes [2], 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). Unfortunately, the AWS Learner Labs provided by AWS do not support ECS.

To resolve this issue, we have created a Terraform module which allows us to deploy Docker images on EC2 instances and abstract over the underlying implementation. The documentation and source for this Terraform module is available on Github: https://github.com/CSSE6400/terraform/tree/main/container.

Using the documentation of the module, combined with the environment variables we know our back-end requires based on the `docker-compose.yml` file, we can develop a resource as below.

```
» cat main.tf
```

```
module "todoapp-backend" {
  source = "git::https://github.com/CSSE6400/terraform//container"

  image = "ghcr.io/csse6400/todo-app:latest"
  instance_type = "t2.micro"
  environment = {
    APP_ENV="local"
    APP_KEY="base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A="
    APP_DEBUG="true"
    LOG_LEVEL="debug"
    DB_CONNECTION="mysql"
    DB_HOST=aws_db_instance.todoapp-database.address
    DB_PORT="3306"
    DB_DATABASE="todoapp"
    DB_USERNAME="todoapp"
    DB_PASSWORD=local.password
  }
  ports = {
    "80" = "8000"
  }
  security_groups = [aws_security_group.todoapp-backend.name]

  tags = {
    Name = "todoapp-backend"
  }
}
```

Note that we are passing the address of our remote database into the container as an environment variable. This is a module which requires a source. In our case, the source will be the Github repository created earlier. Also notice that we map port 80 of the EC2 machine to port 8000 within the container, we should create a security group to make the instance accessible.

```
   » cat main.tf
resource "aws_security_group" "todoapp-backend" {
  name = "todoapp-backend"
  description = "Todo App HTTP and SSH access"

  ingress {
    from_port = 80
    to_port = 80
    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"]
  }
}
```

You will also want to create an output block to expose the address of the instance.

```
    » cat main.tf
output "url" {
  value = module.todoapp-backend.public_dns
}
```

This should give you a `main.tf` file which fully deploys a todo application. If you haven't been applying as we go, try and apply the Terraform file now. If you have any issues, ask your tutor for guidance.

References

[1] M. Kleppmann, *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., March 2017.

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