Storing Stuff

March 14, 2022

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   Working with Docker

So far in the course we have introduced docker as a means to package software to make it easier to work with and deploy. Today we will be using it to run a small application locally that consists of a web server and a relational database.

Figure 2: Sample Todo App made by Brae Webb

> **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.

3.1 Locally

We will be using a container that is built from the Dockerfile described below which can be found here: https://github.com/CSSE6400/todo-app/blob/main/backend/Dockerfile.

```
» cat Dockerfile
1   FROM ubuntu:21.10
2   RUN apt-get update \
3       && DEBIAN_FRONTEND=noninteractive apt install -y \
4           php \
5           php-mysql \
6           php-xml \
7           php-curl \
8           curl \
9           git \
10          unzip
11  RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin
        --filename=composer
12  COPY . /app
13  WORKDIR /app
14  RUN composer install
15  CMD ["php", "artisan", "serve", "--host=0.0.0.0"]
```

Our goal for today is to have a running instance of the Todo App locally 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
1   version: '3.3'
2   services:
3     backend:
4       image: ghcr.io/csse6400/todo-app:latest
5       ports:
```

```
6      - '8000:8000'
7    environment:
8      APP_ENV: 'local'
9      APP_KEY: 'base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A='
10     APP_DEBUG: 'true'
11     LOG_LEVEL: 'debug'
```

A few things to point out in the file. We have defined a single service called backend which uses a pre-made 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.

```
$ 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.

To investigate this error lets hit one of the endpoints for our api. Head over to `http://127.0.0.1:8000/api/v1/todo` which should list all the todos. Once we reach that page we have a clearer idea of whats gone wrong. The page you should see is shown in Figure 3 and the quick summary is that our App is complaining that we haven't given it a database.
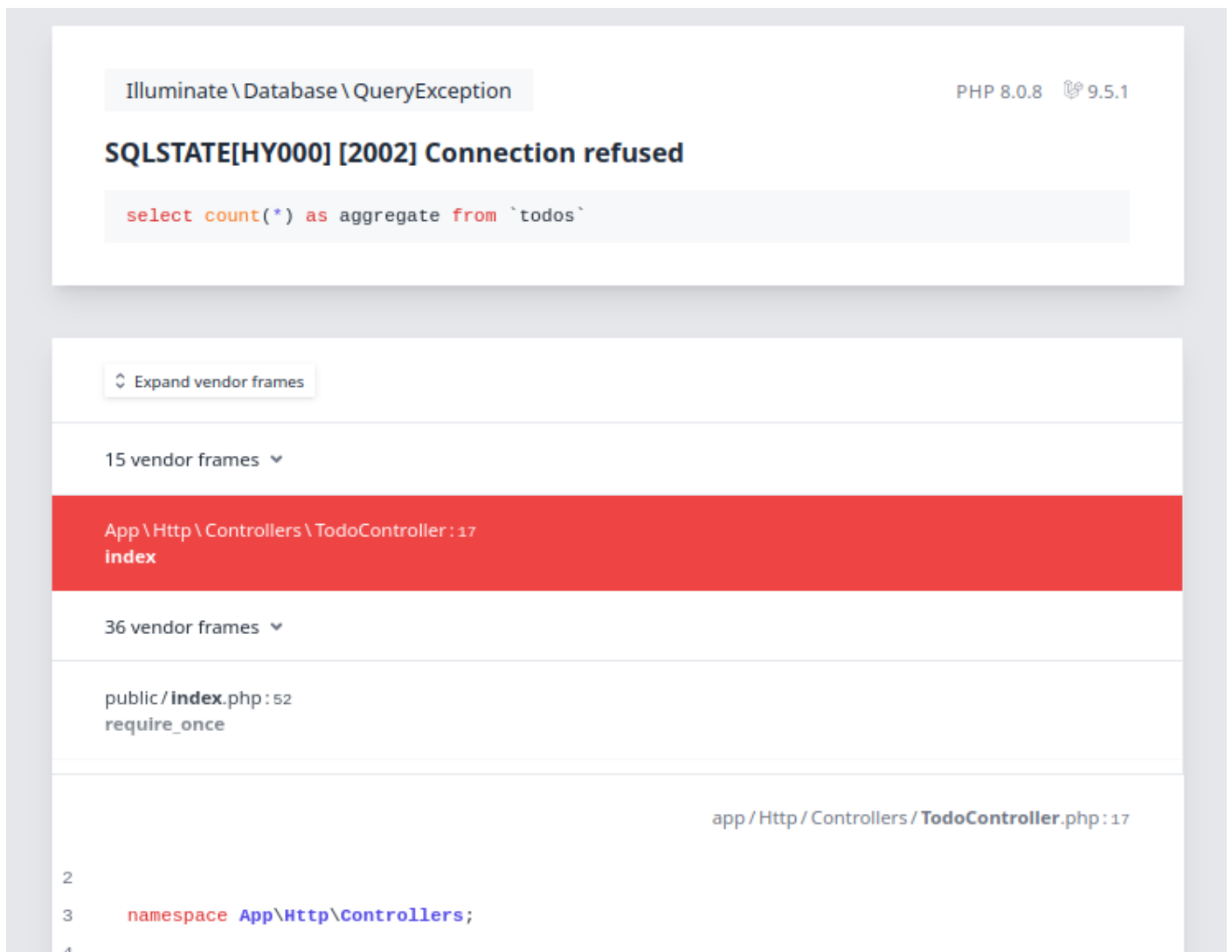
Figure 3: The expected error page when accessing `http://127.0.0.1:8000/api/v1/todo`.

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
1  version: '3.3'
2  services:
3    db:
4      image: mysql:8-debian
5      environment:
6        MYSQL_DATABASE: 'todoapp'
7        MYSQL_USER: 'todoapp'
8        MYSQL_PASSWORD: 'password'
9        MYSQL_ROOT_PASSWORD: 'password'
10     ports:
11       - '3306:3306'

13   backend:
```

```
14      image: ghcr.io/csse6400/todo-app:latest
15      depends_on:
16        - db
17      ports:
18        - '8000:8000'
19      environment:
20        APP_ENV: 'local'
21        APP_KEY: 'base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A='
22        APP_DEBUG: 'true'
23        LOG_LEVEL: 'debug'
24        DB_CONNECTION: 'mysql'
25        DB_HOST: 'db'
26        DB_PORT: '3306'
27        DB_DATABASE: 'todoapp'
28        DB_USERNAME: 'todoapp'
29        DB_PASSWORD: 'password'
```

Now we have two services 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. 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 4. 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.
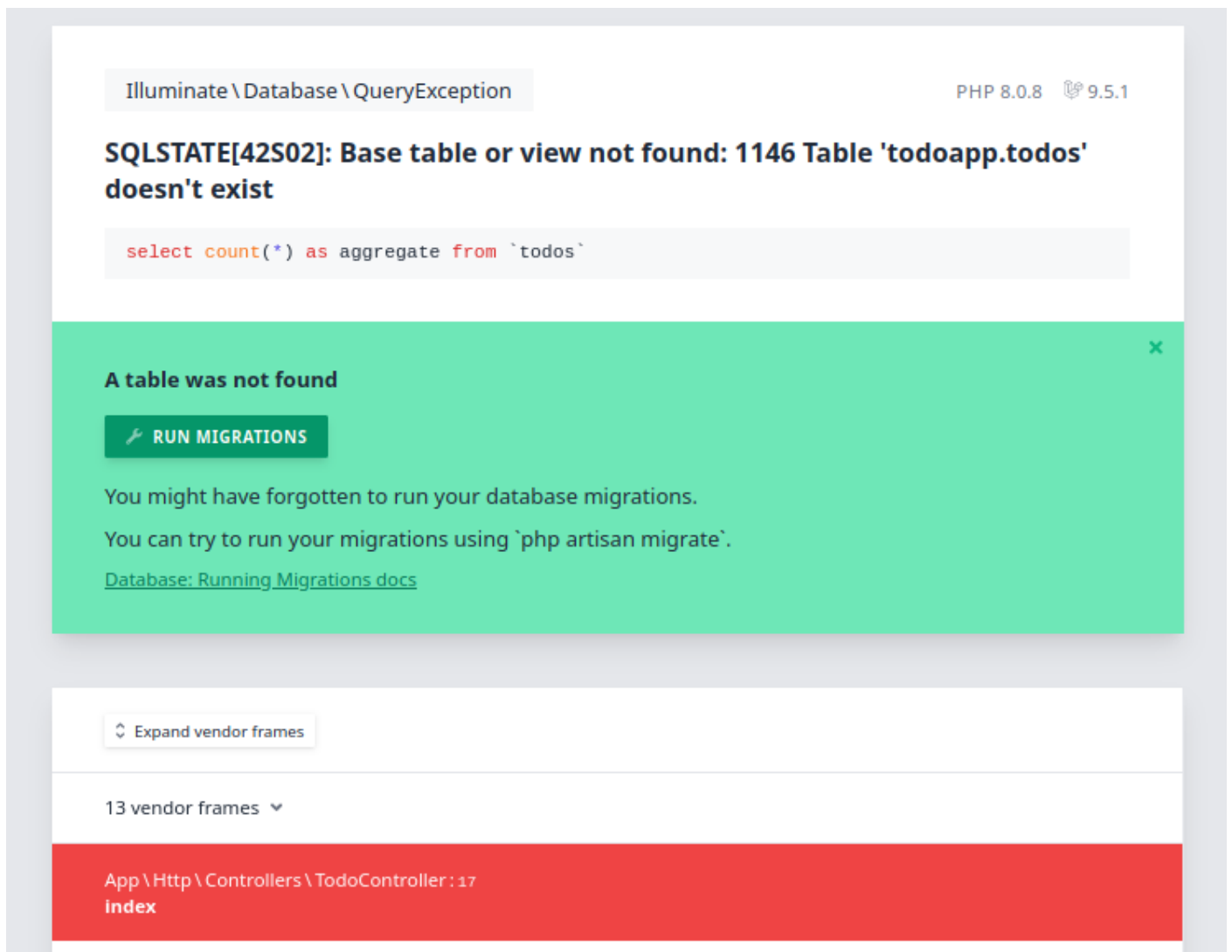
8

Figure 4: 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. Start by opening a new terminal so that we can leave the docker containers running. In this new terminal go to the same directory that we were just in 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 and you should see a fully functional todo app.

My Todo List

Complete CSSE6400 Prac 1	●
Complete CSSE6400 Prac 2	
Complete CSSE6400 Prac 3	
Complete CSSE6400 Prac 4	
Joined the CSSE6400 Slack	
Attended Lecture 1 of CSSE6400	
Attended Lecture 2 of CSSE6400	
Attended Lecture 3 of CSSE6400	
Attended Lecture 4 of CSSE6400	
Attended Braes tutorial	

Previous 1 2 3 Next

> **Info**
>
> These 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.

3.1.1   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
```

```
1  FROM ubuntu:21.10
2  ...
3  ...
4  ...
5  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.

```
1  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
1  version: '3.3'
2  services:
3    db:
4      ...
6    backend:
7      ...
8      environment:
9        ...
10     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.
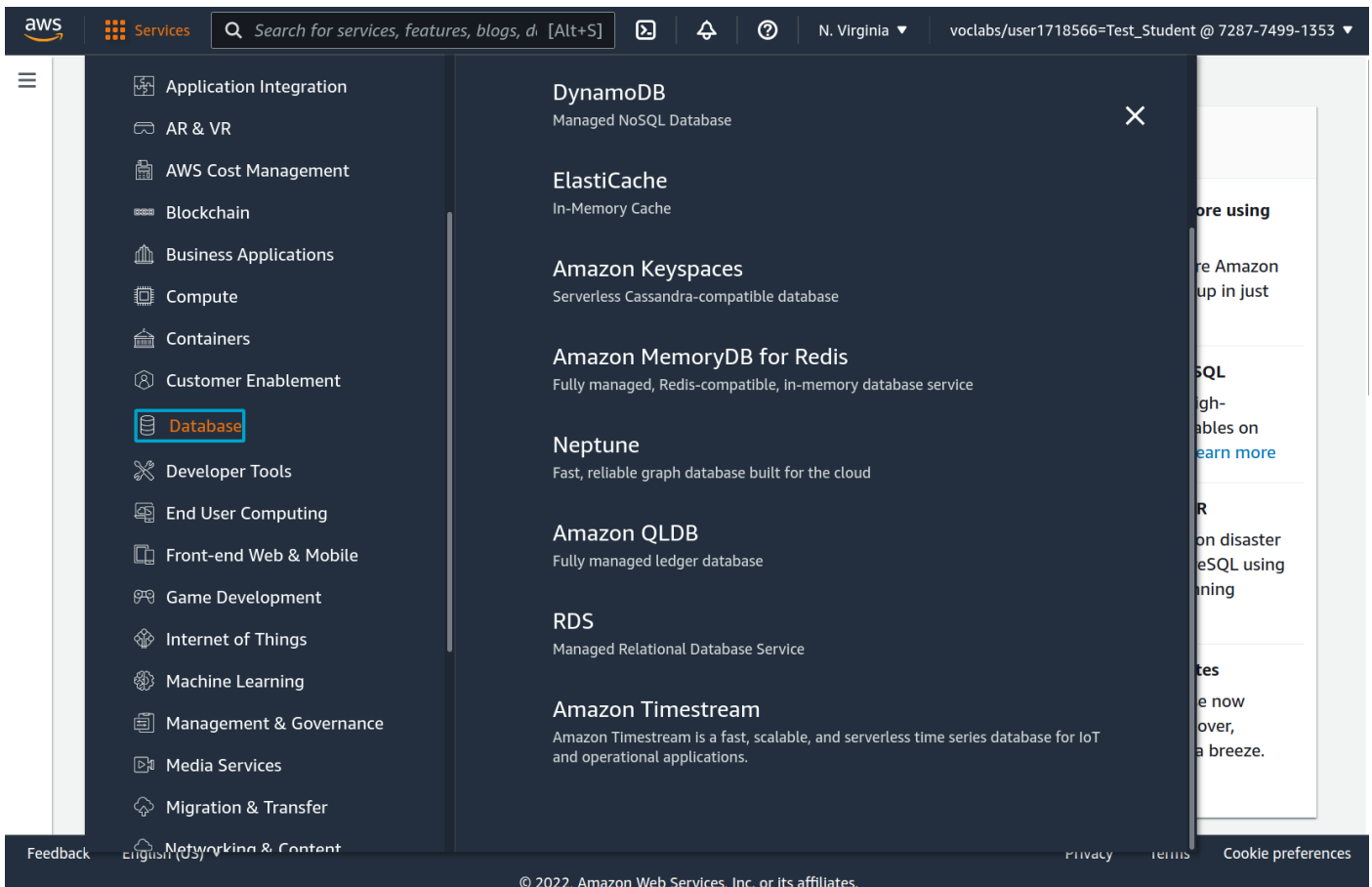
3.2 AWS – Manual

> **Warning**
>
> This section is still being developed.

Now we have a locally running Todo App let's move to AWS. 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)".
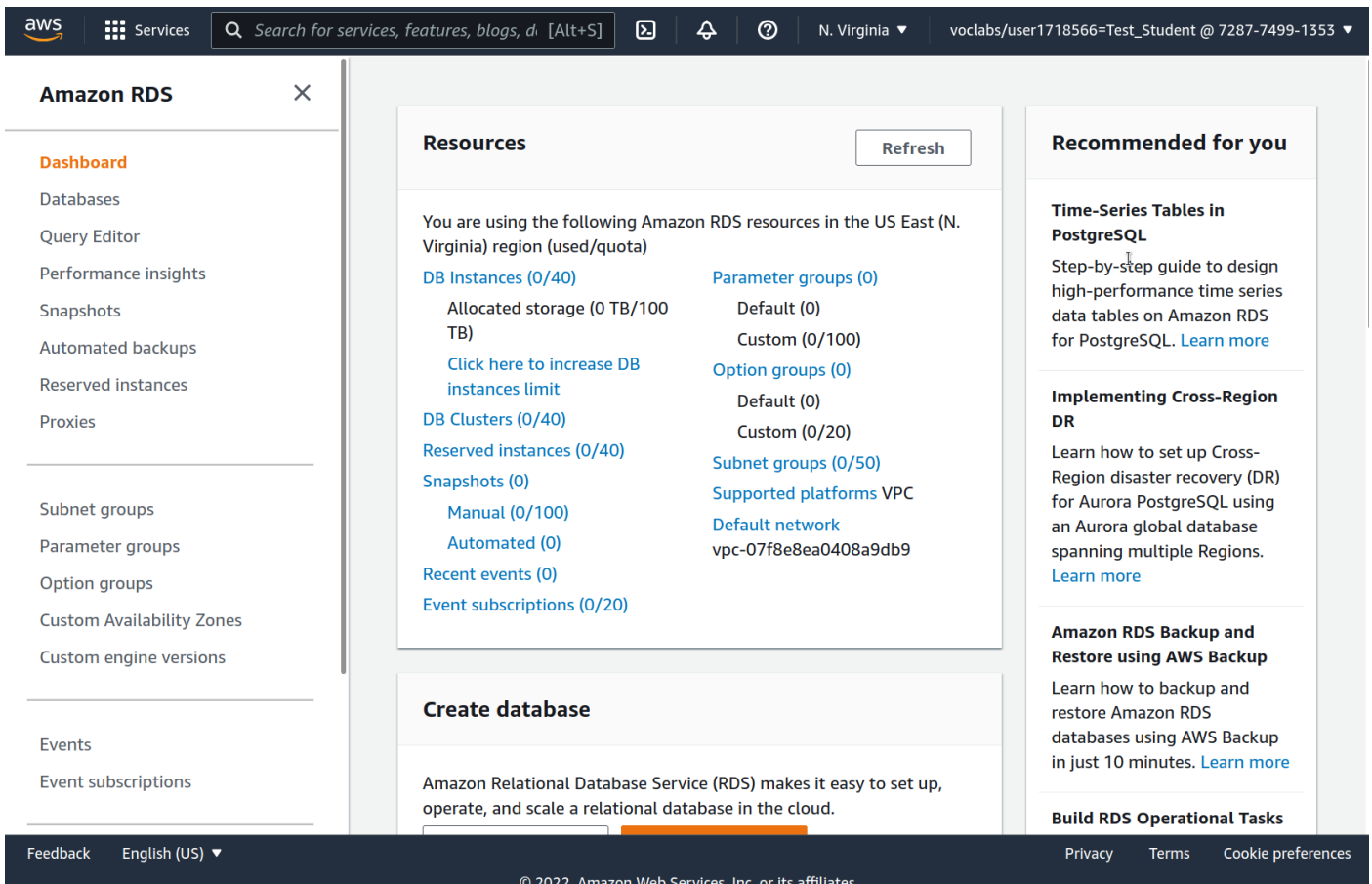
This page should appear familar as it's very similar to the AWS EC2 instance page. Let us create a new database by hitting the "Create Database" button.
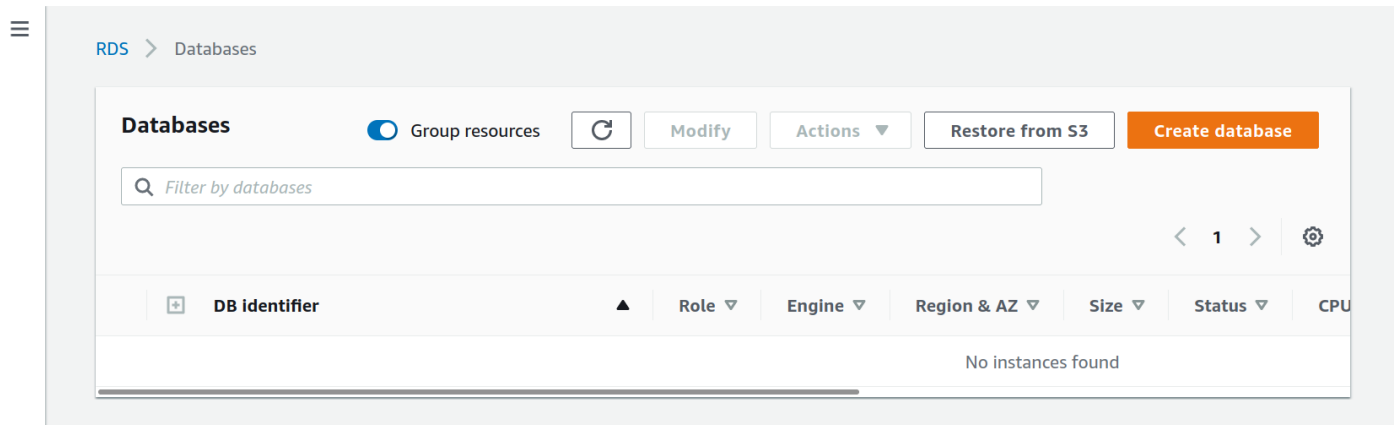
> **Warning**
>
> In the next section we cannot use the Easy Create method as it tries to create a IAM account which is not allowed in the labs. Going forward we would typically do this using terraform so we can easily avoid these restrictions.

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

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

- ○ Amazon Aurora
- ● MySQL
- ○ MariaDB
- ○ PostgreSQL
- ○ Oracle
- ○ Microsoft SQL Server

Edition

- ● MySQL Community

> ⓘ **Known issues/limitations**
> Review the Known issues/limitations ☑ to learn about potential compatibility issues with specific database versions.

Version

    MySQL 8.0.27   ▼

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 connect via. Please enter a reasonable password and keep this aside for later. We will need it for our local docker-compose file.

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.

    todoapp-manual

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.

    todoapp

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

☐ **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 password Info

    ••••••••••••••••••

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

DB instance class

DB instance class Info

○ Standard classes (includes m classes)

○ Memory optimized classes (includes r and x classes)

◉ Burstable classes (includes t classes)

    db.t2.micro
    1 vCPUs   1 GiB RAM   Not EBS Optimized   ▼

◯ 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 don't want to expose your databases publicly and, would instead, have a web server sitting between 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.

Connectivity

Virtual private cloud (VPC) Info
VPC that defines the virtual networking environment for this DB instance.

```
Default VPC (vpc-07f8e8ea0408a9db9)                    ▼
```

Only VPCs with a corresponding DB subnet group are listed.

> ⓘ After a database is created, you can't change its VPC.

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

```
default-vpc-07f8e8ea0408a9db9                          ▼
```

Public access Info

⦿ **Yes**
Amazon EC2 instances and devices outside the VPC can connect to your database. Choose one or more VPC security groups that specify which EC2 instances and devices inside the VPC can connect to the database.

○ **No**
RDS will not assign a public IP address to the database. Only Amazon EC2 instances and devices inside the VPC can connect to your database.

VPC security group
Choose a VPC security group to allow access to your database. Ensure that the security group rules allow the appropriate incoming traffic.

○ **Choose existing**	⦿ **Create new**
Choose existing VPC security groups	Create new VPC security group

New VPC security group name

```
todoapp-manual
```

Availability Zone Info

```
No preference                                          ▼
```

▼ **Additional configuration**

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

```
3306                                                   ⬍
```

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.

Database authentication

Database authentication options Info
- ● **Password authentication**
  Authenticates using database passwords.
- ○ **Password and IAM database authentication**
  Authenticates using the database password and user credentials through AWS IAM users and roles.
- ○ **Password and Kerberos authentication**
  Choose a directory in which you want to allow authorized users to authenticate with this DB instance using Kerberos Authentication.

▼ Additional configuration

Database options, backup enabled, backtrack disabled, Enhanced Monitoring disabled, maintenance, CloudWatch Logs, delete protection disabled.

Database options

Initial database name Info

```
todoapp
```

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

DB parameter group Info

```
default.mysql8.0                                    ▼
```

Option group Info

```
default:mysql-8-0                                   ▼
```

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**

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.

RDS > Databases

Databases   🔵 Group resources   C   Modify   Actions ▼   Restore from S3   **Create database**

🔍 Filter by databases   < 1 >   ⚙

⊞	DB identifier	▲	Role ▽	Engine ▽	Region & AZ ▽	Size ▽	Status ▽	CPU
○	todoapp-manual		Instance	MySQL Community	us-east-1f	db.t2.micro	🕐 Backing-up	▬▭▭10

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.

RDS > Databases > todoapp-manual

todoapp-manual   [Modify] [Actions ▼]

Summary

DB identifier	CPU	Status	Class
todoapp-manual	▪▭ 10.83%	✓ Available	db.t2.micro
Role	Current activity	Engine	Region & AZ
Instance	▭ 0 Connections	MySQL Community	us-east-1f

Connectivity & security | Monitoring | Logs & events | Configuration | Maintenance & backups | Tags

Connectivity & security

Endpoint & port

Endpoint
todoapp-manual.cwf1cdgoxzax.us-east-1.rds.amazonaws.com

Port
3306

Networking

Availability Zone
us-east-1f

VPC
vpc-07f8e8ea0408a9db9

Subnet group
default-vpc-07f8e8ea0408a9db9

Subnets
subnet-0556db549e22800f1
subnet-0da793726639bd6d8
subnet-091ee1d302ae831a9
subnet-0b932c4d6a4154b2a
subnet-0416084227ea643b4
subnet-05d92ccc16a62294f

Security

VPC security groups
todoapp-manual2
(sg-0cc1a6ba52b85e23c)
✓ Active

Publicly accessible
Yes

Certificate authority
rds-ca-2019

Certificate authority date
August 23, 2024, 03:08 (UTC±3:08)

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 host and credentials to match what we input into AWS. Below is an example of my configuration.

```
» cat docker-compose.yml
1  version: '3.3'
2  services:
3    backend:
4      image: ghcr.io/csse6400/todo-app:latest
5      ports:
6        - '8000:8000'
7      environment:
8        APP_ENV: 'local'
9        APP_KEY: 'base64:8PQEPYGlTm1t3aqWmlAw/ZPwCiIFvdXDBjk3mhsom/A='
10       APP_DEBUG: 'true'
11       LOG_LEVEL: 'debug'
12       DB_CONNECTION: 'mysql'
13       DB_HOST: 'todoapp-manual.cwf1cdgoxzax.us-east-1.rds.amazonaws.com'
14       DB_PORT: '3306'
```

```
15        DB_DATABASE: 'todoapp'
16        DB_USERNAME: 'todoapp'
17        DB_PASSWORD: 'MyVerySecurePassword'
18      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.

3.3   AWS - Terraform

Now that we have explored the old school way lets join the 21st century and configure this using IaC.

```
» cat main.tf
1  resource "aws_db_instance" "todoapp-terraform" {
2    allocated_storage = 20
3    max_allocated_storage = 1000
4    engine = "mysql"
5    engine_version = "8.0.27"
6    instance_class = "db.t2.micro"
7    name = "todoapp"
8    username = "todoapp"
9    password = "foobarbaz"
10   parameter_group_name = "default.mysql8.0"
11   skip_final_snapshot = true
12   vpc_security_group_ids = [aws_security_group.todoapp-terraform.id]
13   publicly_accessible = true
14
15   tags = {
```

```
16      Name = "todoapp-terraform"
17    }
18  }

20  resource "aws_security_group" "todoapp-terraform" {
21    name = "todoapp-terraform"
22    description = "Allow inbound MySQL traffic"

24    ingress {
25      from_port = 3306
26      to_port = 3306
27      protocol = "tcp"
28      cidr_blocks = ["0.0.0.0/0"]
29    }

31    egress {
32      from_port = 0
33      to_port = 0
34      protocol = "-1"
35      cidr_blocks = ["0.0.0.0/0"]
36      ipv6_cidr_blocks = ["::/0"]
37    }

39    tags = {
40      Name = "todoapp-terraform"
41    }
42  }
```

> **Warning**
>
> This section is still being developed.

References

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