Workshop 3: Aurora DSQL

Event details

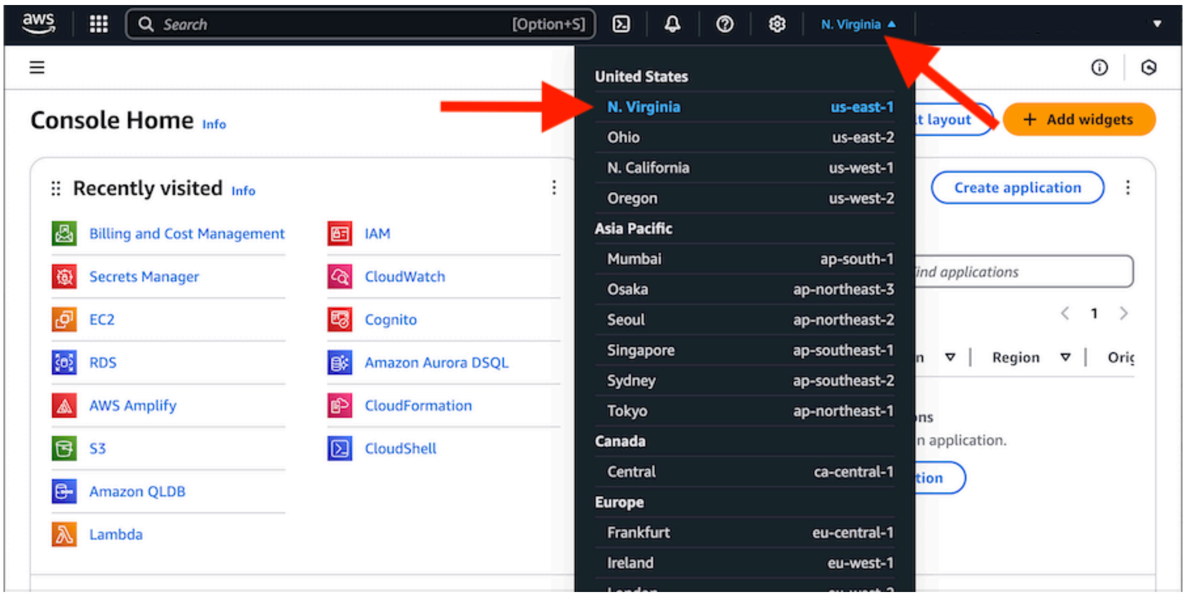
Name
[DAT421] [NEW LAUNCH] Build a multi-Region, active-active rewards app with Amazon Aurora DSQL
Start time
12/06/2024 10:30 AM
Duration
4 hours
Level
400
Description
In this workshop, get hands-on experience with Amazon Aurora DSQL, a new serverless distributed SQL database that delivers active-active high availability. Learn how to build a retail rewards points application with active-active resiliency across multiple Availability Zones and two Regions.

AWS CLI Login:
export AWS_DEFAULT_REGION="us-east-1"
export AWS_ACCESS_KEY_ID="ASIA42BEEJZHAMNQOLV2"
export AWS_SECRET_ACCESS_KEY="/
HqKg7Y5iDIAZbrnNpD5B3X0hNLrCPIUKurCS3ci"
export
AWS_SESSION_TOKEN="IQoJb3JpZ2luX2VjEHsaCXVzLWVhc3QtMSJIMEYCIQ
Dx/
d+I97qqz6m7pzGdY34qPXf4SCJLtwrmqHL175cE7wIhAMS87kj0YtM2YH+Nu/
16j5HzBwkmJQSe7oocqKI8zDnmKpkCCDQQARoMODgwNTQ0MDc1MzQyIgxT
mA/
JscoyyfR72cMq9gGT4+kyCFlOhbsUuIe5V6bUe4TpDzvmcqvCLAhkUWWOFhi/
SHCDBadb78y2muRgfdxgudxoot+TwPz6MQ2IBOrDMZU2xjwWZjkPfmft5r7v0M
CQhthVGJem/CQvxHq75vcUMGsfKvbOa3nyAKtUoCGve/
s74/4zrEzXeicguIsmuPvGuVQeNE4nx0c7/
PshFyfFsrk1Ptx8Jj7nwW3UeSs3mz+KpFGxq/J1rsCgkkAH1PCeyGR/
XUrogX6SKtQTnJ9vjvJPRdcHOukydfn7lofW2cpATEoqQDeSQj5vJkO3z0ImPtXJ
66psbCSbDTCpuwT5EBOC86QwpIfNugY6nAGR7xvDxOjEAJhY/
9GEkI+VWFfmRGvdAhyaVtdJdU0XC3NcdwZ5m4JIeja/
pQ2eeV+JES+7ewu0B2TpA6EY2L6UPSYJjJOHKOncyd8o/
D9vO4vk991eb8Sr+Z45n2zBk5UC/
1OhP4jqHOyKXgV8weP+op5+IigbJZSbSEwh9QwuL6l01tc4giW/
JCbDQnvpEhpTUSVHBhu5rw4kSYY="

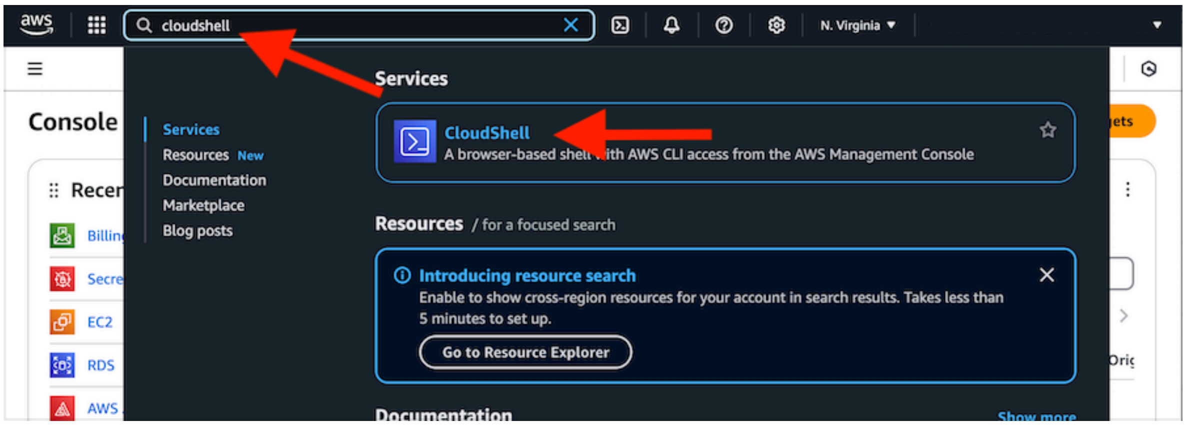
Setup first region

In this module, we'll setup CloudShell in our first region: us-east-1 (Northern Virginia).
In the AWS Management Console, check the currently-selected

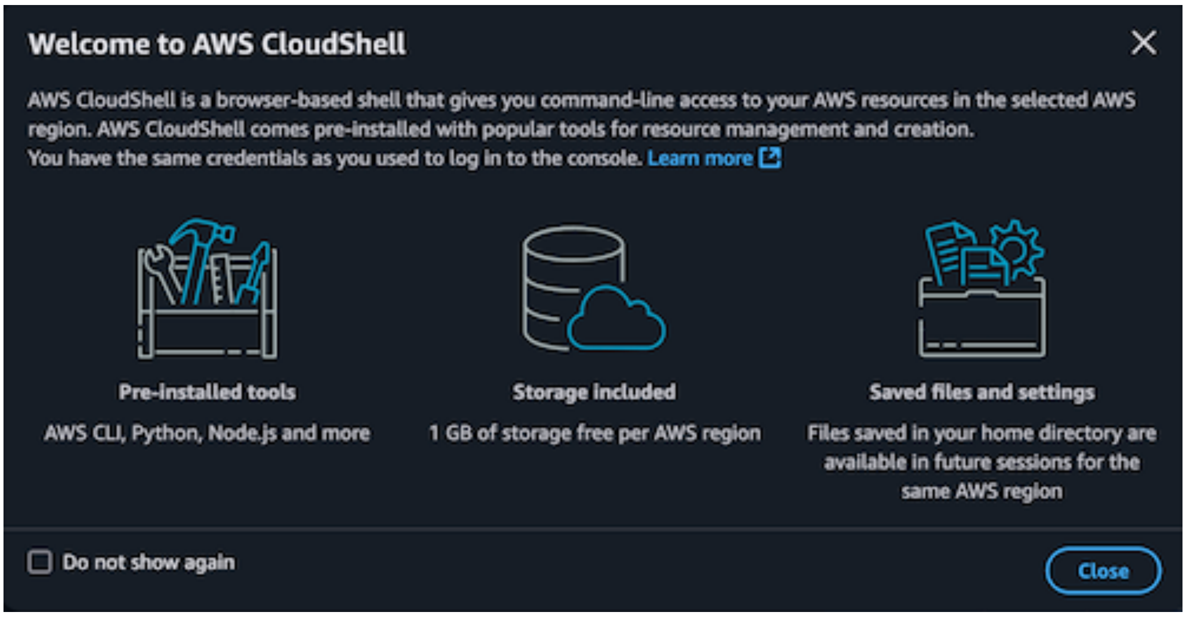
region in the menu bar at the top of the page. If the region is not set to "N. Virginia", click the down arrow next to the current region and select "us-east-1 N. Virginia" from the drop-down menu.



In the search bar at the top of the console, enter "cloudshell" and click the "CloudShell" link to open a new CloudShell terminal session.


aws
cloudshell
N. Virginia
Console
Services
Resources New
Documentation
Marketplace
Blog posts
Services
CloudShell
A browser-based shell with AWS CLI access from the AWS Management Console
Resources / for a focused search
Introducing resource search
Enable to show cross-region resources for your account in search results. Takes less than 5 minutes to set up.
Go to Resource Explorer
Documentation
Show more

You may be presented with a "Welcome to AWS CloudShell" dialog. If so, check the Do not show again box and click Close.



Welcome to AWS CloudShell

AWS CloudShell is a browser-based shell that gives you command-line access to your AWS resources in the selected AWS region. AWS CloudShell comes pre-installed with popular tools for resource management and creation.
You have the same credentials as you used to log in to the console. Learn more

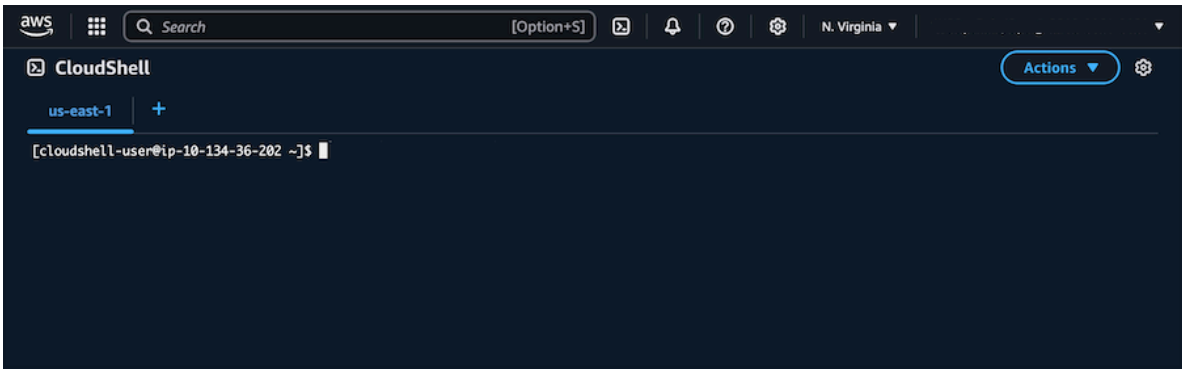
Pre-installed tools
AWS CLI, Python, Node.js and more

Storage included
1 GB of storage free per AWS region

Saved files and settings
Files saved in your home directory are available in future sessions for the same AWS region

☐ Do not show again

CloudShell will initialize a terminal session and, after a few moments, present you with a Linux shell prompt.

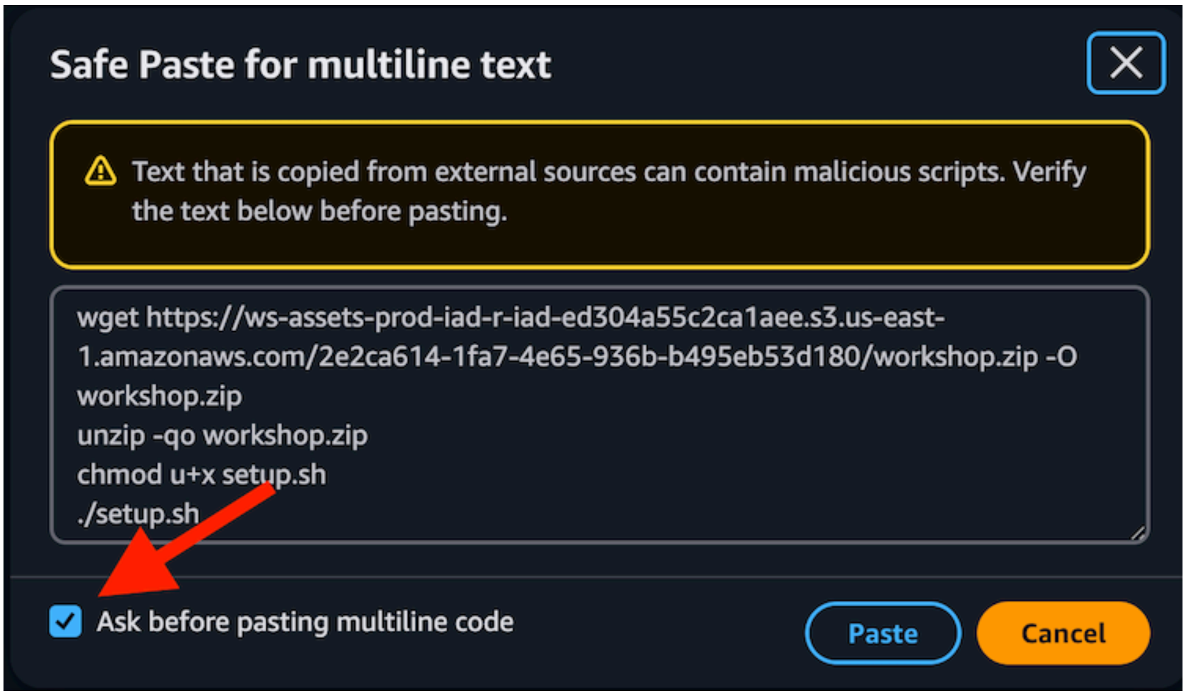


```
[cloudshell-user@ip-10-134-36-202 ~]$
```

Next, we'll install a Java Development Kit, the Apache Maven build automation tool, and other dependencies using a setup script. Download the setup script to your CloudShell session by executing the commands below from CloudShell.

wget https://ws-assets-prod-iad-r-iad-ed304a55c2ca1aee.s3.us-east-1.amazonaws.com/2e2ca614-1fa7-4e65-936b-b495eb53d180/workshop.zip -O workshop.zip
unzip -qo workshop.zip
chmod u+x setup.sh
./setup.sh

You may be presented with the "Safe Paste for multiline text" dialog. If so, Uncheck the Ask before pasting multiline code box and click Paste.



Text that is copied from external sources can contain malicious scripts. Verify the text below before pasting.

```
wget https://ws-assets-prod-iad-r-iad-ed304a55c2ca1aee.s3.us-east-1.amazonaws.com/2e2ca614-1fa7-4e65-936b-b495eb53d180/workshop.zip -O workshop.zip
unzip -qo workshop.zip
chmod u+x setup.sh
./setup.sh
```

[✓] Ask before pasting multiline code

Now run the following command to update your environment's settings.

```
. ./.bashrc
```

Verify that our dependencies are installed. Run java --version in CloudShell. You should see Java version 21 or higher installed.

```
[cloudshell-user@ip-xx-xxx-xx-xxx ~]$ java --version
openjdk 21.0.5 2024-10-15 LTS
OpenJDK Runtime Environment Corretto-21.0.5.11.1 (build 21.0.5+11-LTS)
OpenJDK 64-Bit Server VM Corretto-21.0.5.11.1 (build 21.0.5+11-LTS, mixed mode, sharing)
```

Run mvn --version in CloudShell. You should see Apache Maven version 3.9.9 or higher installed.

```
[cloudshell-user@ip-xx-xxx-xx-xxx ~]$ mvn --version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: /home/cloudshell-user/apache-maven-3.9.9
Java version: 21.0.5, vendor: Amazon.com Inc., runtime: /usr/lib/jvm/java-21-amazon-corretto.x86_64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "6.1.112-124.190.amzn2023.x86_64", arch: "amd64", family: "unix"
```

The CloudShell environment is now ready. In the next lesson, we'll setup CloudShell in our second region.

Handling CloudShell Timeouts

CloudShell sessions end after 20-30 minutes of inactivity, which may occur while following the steps in this workshop. If you see a

timeout message or if commands produce error messages or don't seem to run as expected, re-initialize the shell session by running the following commands:

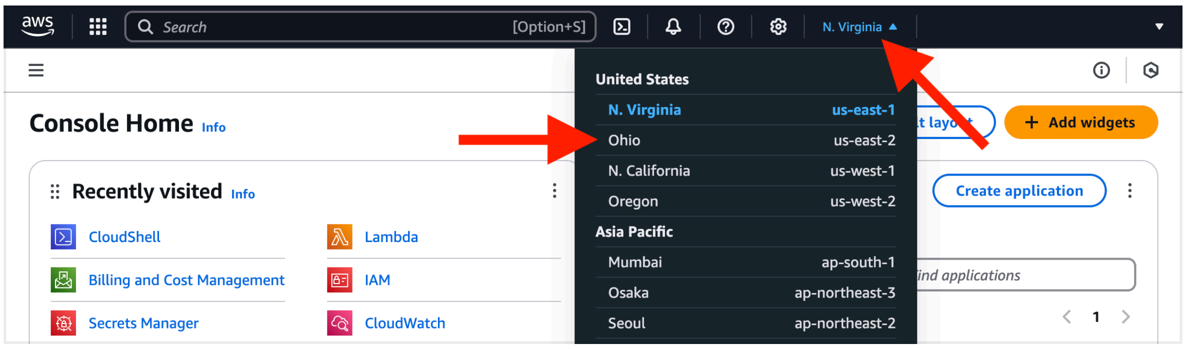
```
./setup.sh
. ./.bashrc
```

Setup second region

In this module, we'll setup CloudShell in our second region: us-east-2 (Ohio).

Open the AWS Management Console in a second browser tab or window. In most web browsers, you can do this by right-clicking the AWS logo in the upper left-hand corner of the console and clicking "Open Link in New Tab" or "Open Link in New Window".

In the second tab, click the down arrow next to "N. Virginia" in the menu bar and select the "Ohio us-east-2" region from the drop-down menu.



You should now have a browser tab or window for us-east-1 (Northern Virginia) and one for us-east-2 (Ohio).

In the us-east-2 tab, open a new CloudShell terminal session just as you did in Setup first region. Run the commands below in the CloudShell terminal.

```
wget https://ws-assets-prod-iad-r-iad-ed304a55c2ca1aee.s3.us-east-1.amazonaws.com/2e2ca614-1fa7-4e65-936b-b495eb53d180/workshop.zip -O workshop.zip
unzip -qo workshop.zip
chmod u+x setup.sh
./setup.sh
```

Now run the following command to update your environment's settings.

```
. ./.bashrc
```

CloudShell environments in both regions are now ready.
Congrats! You have performed all of the necessary setup instructions. Leave both browser tabs or windows open as you work through the remaining modules of the workshop.
Click Next to begin working with Aurora DSQL.

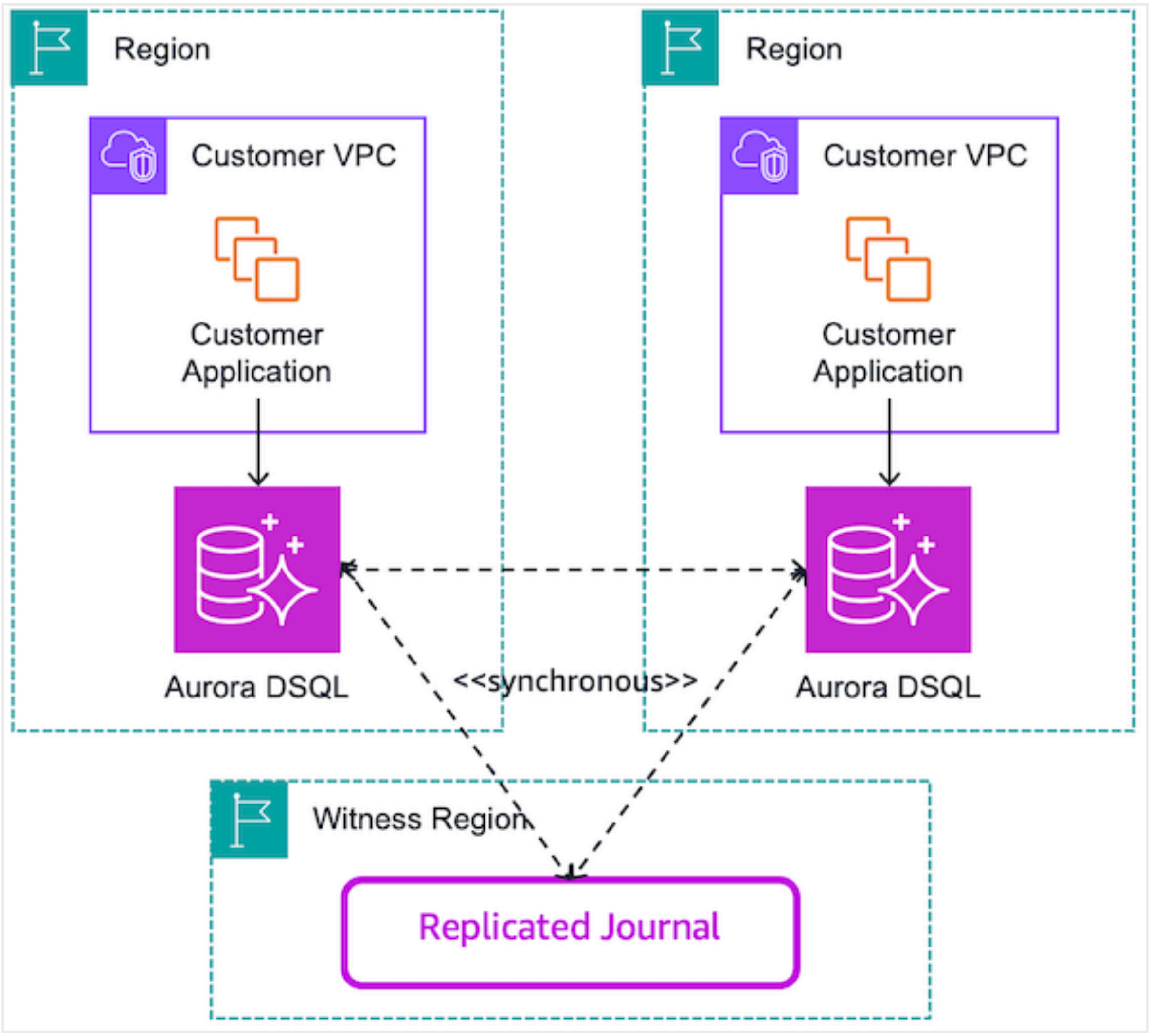
Handling CloudShell Timeouts

CloudShell sessions end after 20-30 minutes of inactivity, which may occur while following the steps in this workshop. If you see a timeout message or if commands produce error messages or don't seem to run as expected, re-initialize the shell session by running the following commands:

```
./setup.sh
. ./.bashrc
```

2. Working with Aurora DSQL

Amazon Aurora DSQL is a new serverless distributed SQL database, which supports active-active configurations in both single and multi-region clusters. In a single-region cluster, the data is replicated across 3 AZs, with 99.99% availability. In a multi-region cluster, the database is region-level fault tolerant: this means 99.999% availability. Aurora DSQL multi-region clusters offer two regional endpoints that support strongly consistent reads and writes, and a third region (called a witness region) that cannot be accessed. The witness region is used for write quorum and acts as a tie breaker to decide which region gets to accept writes in the event of a network partition. The witness region contains a copy of the transaction journal but not the full database.
Below is an architecture diagram of a multi-region Aurora DSQL cluster:



In this module, we will create a multi-region Aurora DSQL cluster, create the IAM roles to connect to it, load and query sample data, and explore the data model and transactional best practices.
Let's start by creating an Aurora DSQL cluster. Click **Next** to continue

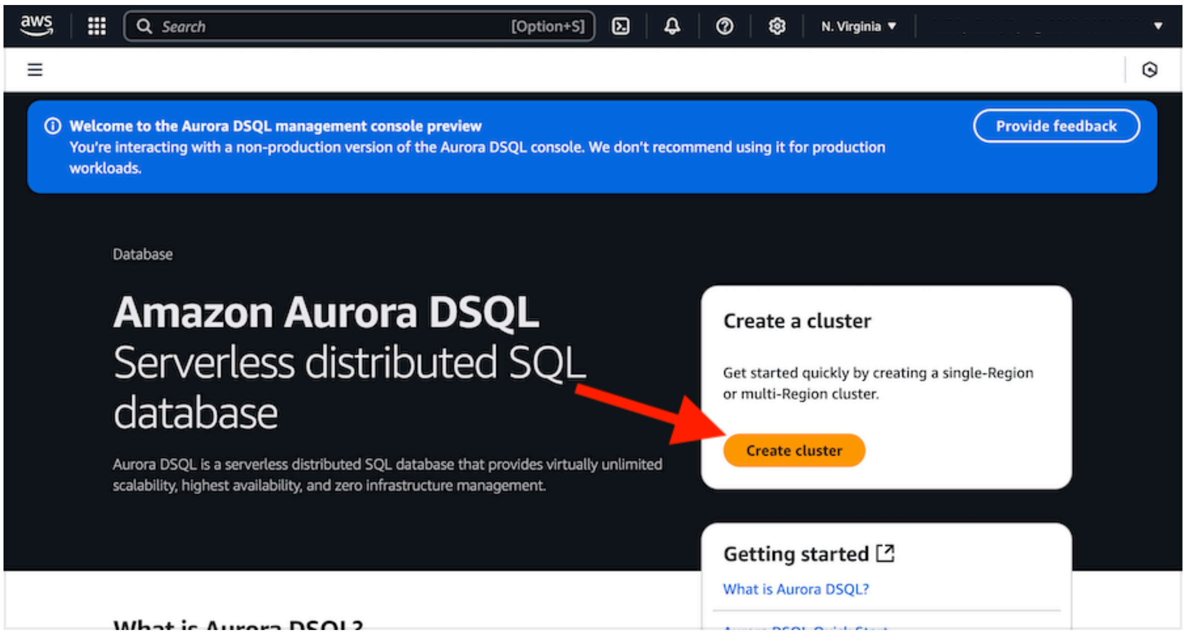
Create a multi-region Aurora DSQL cluster

In this module, we'll create a multi-region Amazon Aurora DSQL cluster.
The workshop environment setup has already granted the privileges required to create, manage and connect to Aurora DSQL clusters with admin privileges.
This is what an IAM policy with the required privileges looks like:

```
{
  "Version" : "2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "dsql:CreateCluster",
        "dsql:GetCluster",
                "dsql:UpdateCluster",
                "dsql:DeleteCluster",
                "dsql:ListClusters",
                "dsql:CreateMultiRegionClusters",
                "dsql:DeleteMultiRegionClusters",
                "dsql:DbConnect",
                "dsql:DbConnectAdmin"
            ],
            "Resource" : "*"
        }
    ]
}
```
To create a new Aurora DSQL cluster, sign into the AWS Management Console: https://us-east-1.console.aws.amazon.com/console/

Then, change your URL to set dsql as the path: https://us-east-1.console.aws.amazon.com/dsql/

.
From the Aurora DSQL welcome page, select **Create Cluster**.



Welcome to the Aurora DSQL management console preview
You're interacting with a non-production version of the Aurora DSQL console. We don't recommend using it for production workloads.

Database

Amazon Aurora DSQL
Serverless distributed SQL database

Aurora DSQL is a serverless distributed SQL database that provides virtually unlimited scalability, highest availability, and zero infrastructure management.

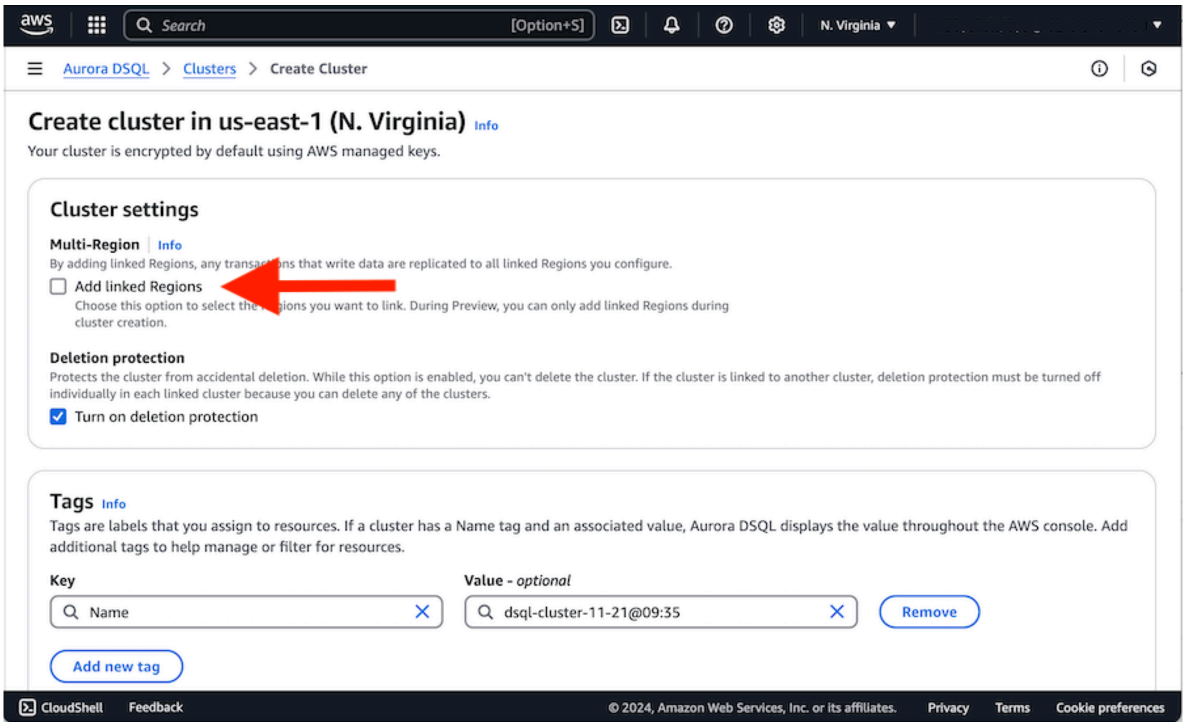
Create a cluster

Get started quickly by creating a single-Region or multi-Region cluster.

Getting started

What is Aurora DSQL?

Check the **Add linked Regions** box.



Create cluster in us-east-1 (N. Virginia) Info

Your cluster is encrypted by default using AWS managed keys.

Cluster settings

Multi-Region | Info

By adding linked Regions, any transactions that write data are replicated to all linked Regions you configure.

☐ Add linked Regions

Choose this option to select the Regions you want to link. During Preview, you can only add linked Regions during cluster creation.

Deletion protection

Protects the cluster from accidental deletion. While this option is enabled, you can't delete the cluster. If the cluster is linked to another cluster, deletion protection must be turned off individually in each linked cluster because you can delete any of the clusters.

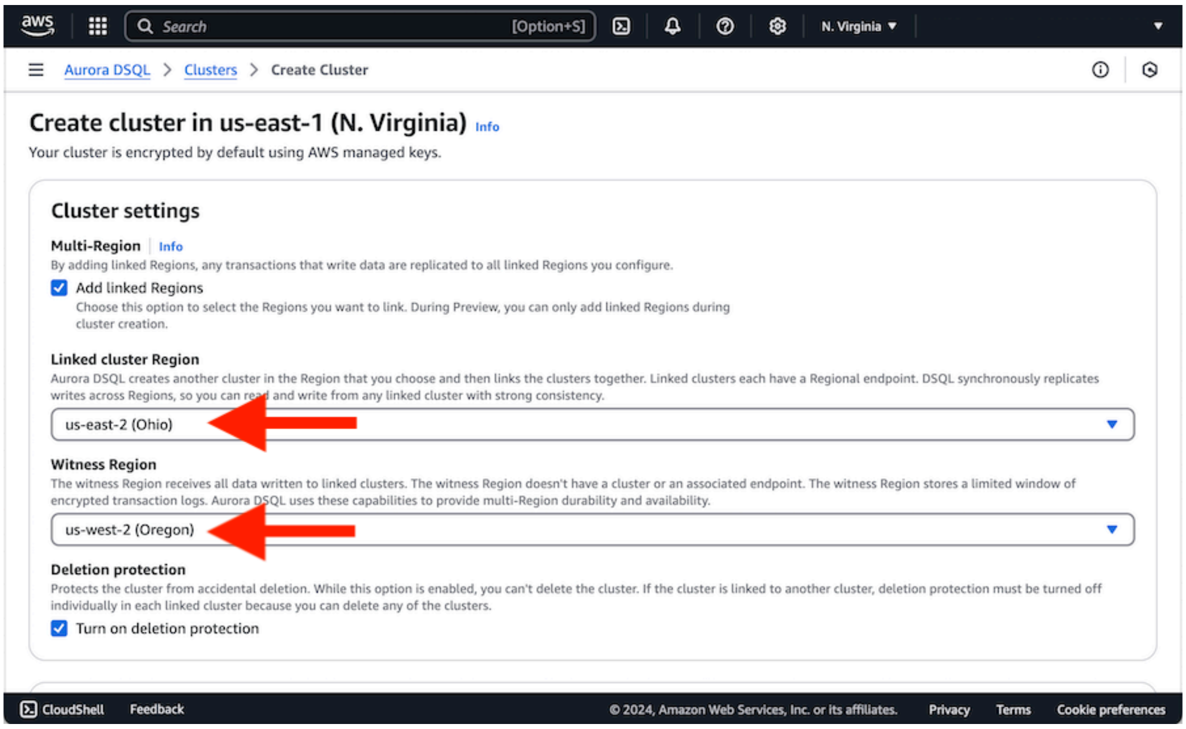
☑ Turn on deletion protection

Tags Info

Tags are labels that you assign to resources. If a cluster has a Name tag and an associated value, Aurora DSQL displays the value throughout the AWS console. Add additional tags to help manage or filter for resources.

Key	Value - *optional*	
Name	dsql-cluster-11-21@09:35	Remove

Choose **us-east-2 (Ohio)** for the **Linked cluster Region**, and **us-west-2 (Oregon)** for the **Witness Region**. Enter "Rewards" for the "Name" tag and click **Create Cluster**.



Create cluster in us-east-1 (N. Virginia) Info

Your cluster is encrypted by default using AWS managed keys.

Cluster settings

Multi-Region | Info

By adding linked Regions, any transactions that write data are replicated to all linked Regions you configure.

☑ Add linked Regions

Choose this option to select the Regions you want to link. During Preview, you can only add linked Regions during cluster creation.

Linked cluster Region

Aurora DSQL creates another cluster in the Region that you choose and then links the clusters together. Linked clusters each have a Regional endpoint. DSQL synchronously replicates writes across Regions, so you can read and write from any linked cluster with strong consistency.

us-east-2 (Ohio)

Witness Region

The witness Region receives all data written to linked clusters. The witness Region doesn't have a cluster or an associated endpoint. The witness Region stores a limited window of encrypted transaction logs. Aurora DSQL uses these capabilities to provide multi-Region durability and availability.

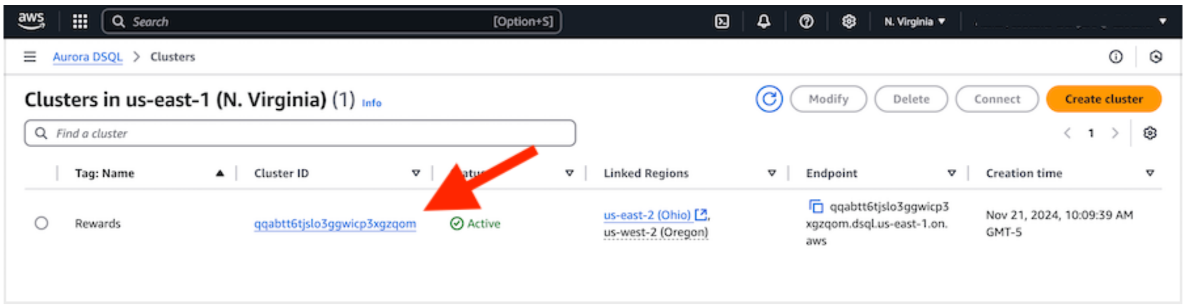
us-west-2 (Oregon)

Deletion protection

Protects the cluster from accidental deletion. While this option is enabled, you can't delete the cluster. If the cluster is linked to another cluster, deletion protection must be turned off individually in each linked cluster because you can delete any of the clusters.

☑ Turn on deletion protection

When the cluster creation completes, click its link in the list of clusters to see details about the cluster.

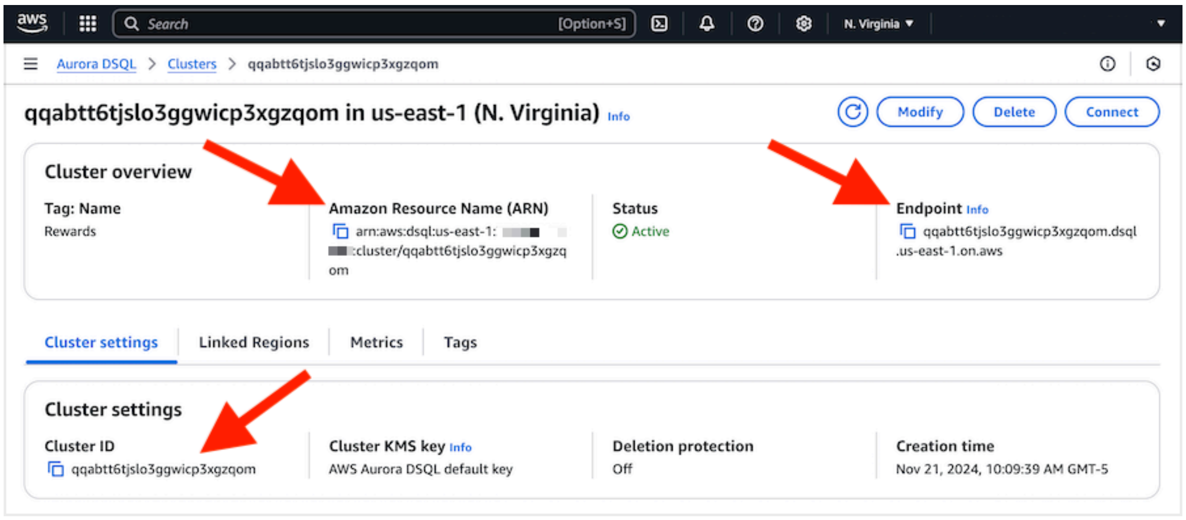


Aurora DSQL > Clusters

Clusters in us-east-1 (N. Virginia) (1) Info

	Tag: Name	Cluster ID	atu	Linked Regions	Endpoint	Creation time
○	Rewards	qqabtt6tjslo3ggwicp3xgzqom	Active	us-east-2 (Ohio), us-west-2 (Oregon)	qqabtt6tjslo3ggwicp3xgzqom.dsql.us-east-1.on.aws	Nov 21, 2024, 10:09:39 AM GMT-5

The details page contains useful information that we'll need throughout this workshop, including the **Cluster ID**, the cluster **Endpoint**, and the cluster **Amazon Resource Name (ARN)**. In a multi-region cluster like the one we just created, the linked cluster in the other region will have its own **Cluster ID**, **Endpoint**, and **Cluster ARN**. You'll need the ID, endpoint, and ARN for whichever regional cluster you want to connect to.



Aurora DSQL > Clusters > qqabtt6tjslo3ggwicp3xgzqom

qqabtt6tjslo3ggwicp3xgzqom in us-east-1 (N. Virginia) Info

Cluster overview

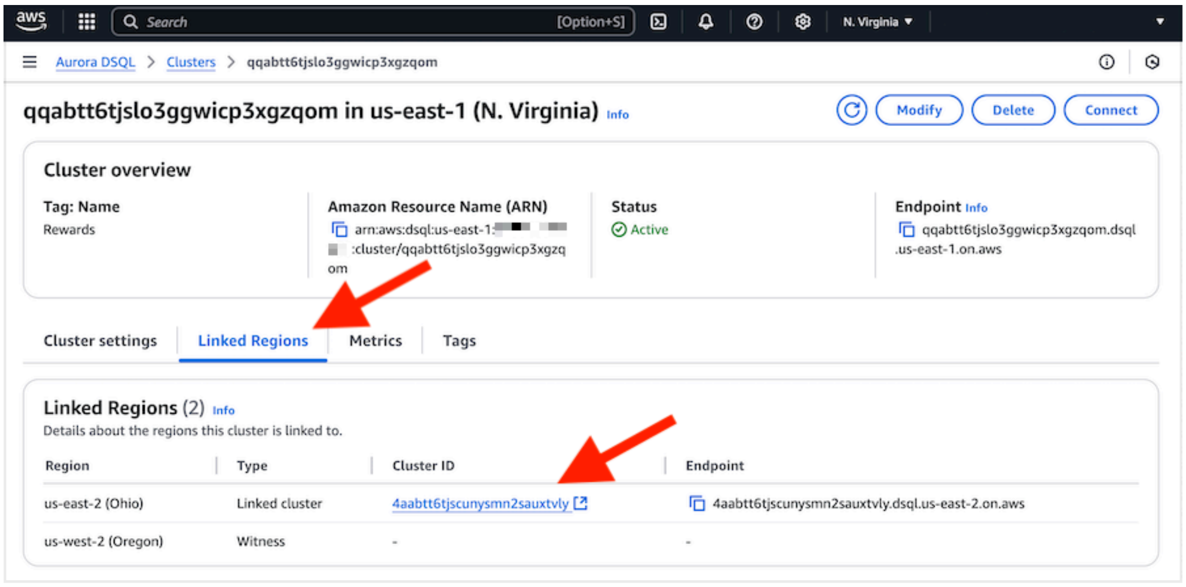
Tag: Name	Amazon Resource Name (ARN)	Status	Endpoint Info
Rewards	arn:aws:dsql:us-east-1:[redacted]:cluster/qqabtt6tjslo3ggwicp3xgzqom	Active	qqabtt6tjslo3ggwicp3xgzqom.dsql.us-east-1.on.aws

Cluster settings | Linked Regions | Metrics | Tags

Cluster settings

Cluster ID	Cluster KMS key Info	Deletion protection	Creation time
qqabtt6tjslo3ggwicp3xgzqom	AWS Aurora DSQL default key	Off	Nov 21, 2024, 10:09:39 AM GMT-5

To see the information for the clusters this one is linked to, click the **Linked Regions** tab. You'll see the Witness Region resource and the ID of the linked cluster in the other read-write region. Click the link for the **Linked cluster** in the other read-write region from the **Linked Cluster ID** column of the **Linked Clusters** table.



Aurora DSQL > Clusters > qqabtt6tjslo3ggwicp3xgzqom

qqabtt6tjslo3ggwicp3xgzqom in us-east-1 (N. Virginia)

Cluster overview

Tag: Name	Amazon Resource Name (ARN)	Status	Endpoint
Rewards	arn:aws:dsql:us-east-1:[redacted]:cluster/qqabtt6tjslo3ggwicp3xgzqom	Active	qqabtt6tjslo3ggwicp3xgzqom.dsql.us-east-1.on.aws

Linked Regions (2)

Details about the regions this cluster is linked to.

Region	Type	Cluster ID	Endpoint
us-east-2 (Ohio)	Linked cluster	4aabtt6tjscunysmn2sauxtvly	4aabtt6tjscunysmn2sauxtvly.dsql.us-east-2.on.aws
us-west-2 (Oregon)	Witness	-	-

This brings you to the details page for the cluster in the other region. Click **Next** to proceed to the next section where we'll connect to Aurora DSQL as the database admin user.

Connect to Aurora DSQL as admin

In this module we will connect to the Aurora DSQL cluster you just created, and we will learn more about connectivity in Aurora DSQL.

To connect as the database admin user, you must use an IAM identity that has access to the IAM policy action dsql:DbConnectAdmin. For example, the below policy allows an IAM role to access as DbConnectAdmin all the Aurora DSQL clusters created in the us-east-2 region in the 111122223333 AWS account. You will not need to create this. The workshop environment setup already created an Aurora DSQL admin user with all the privileges required to connect as admin to the Aurora DSQL clusters in this account.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AuroraDSQLDatabaseConnect",
      "Effect": "Allow",
      "Action": [
        "dsql:DbConnectAdmin"
      ],
      "Resource": "arn:aws:dsql:us-east-2:111122223333:cluster/*"
    }
  ]
}
```

All we need to do now is to use this policy and authenticate as admin to our Aurora DSQL cluster.

Aurora DSQL does not store long-lived credentials. Passwords in Aurora DSQL are temporary IAM authentication tokens that you can generate using the AWS command-line interface (CLI) or software development kits (SDKs).

Connect to Aurora DSQL in us-east-1

Return to the CloudShell browser tab or window for us-east-1 that you opened in the Getting Started module of this workshop. Run the code below in CloudShell, replacing <<YOUR_CLUSTER_ENDPOINT>> with the us-east-1 endpoint of the Aurora DSQL cluster that you created in Create a multi-region Aurora DSQL cluster.

```
export CLUSTER_ENDPOINT=<<YOUR_CLUSTER_ENDPOINT>>
```

Now run the following code:

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --
hostname $CLUSTER_ENDPOINT --expires_in 14400)
export DBNAME="postgres"
export USER="admin"
export HOST=$CLUSTER_ENDPOINT
```

This creates an authentication token, storing it in the PGPASSWORD environment variable. The default expiration for authentication tokens is 15 minutes (900 seconds). We've set ours to expire in 4 hours (14,400 seconds) for this workshop. The code also sets up a few other environment variables that we'll need.

Now that we have all the necessary connection parameters, we can attempt a connection to our cluster.

```
psql --dbname $DBNAME --host $HOST --username $USER --
set=sslmode=require
```

Note that for sslmode, Aurora DSQL does not support verify-full. Aurora DSQL does support allow, prefer, require and verify-ca.

```
[cloudshell-user@ip-xx-xxx-xx-xxx aws]$ export
CLUSTER_ENDPOINT=qqabtt6tjslo3ggwicp3xgzqom.dsql.us-east-1.on.aws
[cloudshell-user@ip-xx-xxx-xx-xxx aws]$ export PGPASSWORD=$(aws dsql
generate-db-connect-admin-auth-token --hostname $CLUSTER_ENDPOINT --
expires_in 14400)
[cloudshell-user@ip-xx-xxx-xx-xxx aws]$ export DBNAME="postgres"
[cloudshell-user@ip-xx-xxx-xx-xxx aws]$ export USER="admin"
[cloudshell-user@ip-xx-xxx-xx-xxx aws]$ export HOST=$CLUSTER_ENDPOINT
```

```
[cloudshell-user@ip-xx-xxx-xx-xxx aws]$ psql --dbname $DBNAME --host
$HOST --username $USER --set=sslmode=require
psql (15.8, server 16.5)
WARNING: psql major version 15, server major version 16.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_128_GCM_SHA256,
compression: off)
Type "help" for help.
postgres=>
```

Connect to Aurora DSQL in us-east-2

Switch to the browser tab or window for us-east-2. You should have an open CloudShell terminal session in us-east-2 from Setup second region. Run the code below in CloudShell, replacing <<YOUR_CLUSTER_ENDPOINT>> with the us-east-2 endpoint of the Aurora DSQL cluster that you created in Create a multi-region Aurora DSQL cluster.

```
export CLUSTER_ENDPOINT=<<YOUR_CLUSTER_ENDPOINT>>
```

Now run the following code:

```
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token --
hostname $CLUSTER_ENDPOINT --expires_in 14400)
export DBNAME="postgres"
export USER="admin"
export HOST=$CLUSTER_ENDPOINT
```

This is the same code that we ran in us-east-1, but we've changed the cluster endpoint for us-east-2. Now connect to Aurora DSQL as before by running:

```
psql --dbname $DBNAME --host $HOST --username $USER --
set=sslmode=require
```

Database Connection Timeouts

Aurora DSQL connections timeout after 60 minutes duration, so you may need to reconnect to Aurora DSQL several times during the workshop. If your session times out, you'll see a message like this:

```
SSL SYSCALL error: EOF detected
The connection to the server was lost. Attempting reset: Failed.
!?>
```

To reconnect your session, exit psql by running \q, then reconnect as usual.

That's it! You should now have an authenticated psql session into Aurora DSQL in both us-east-1 and us-east-2. In the next section,

we'll create our database objects and load our application's sample data. Click Next to continue.

Create schema and load data

In this module we will create objects and load data into the Aurora DSQL cluster.

Amazon Aurora DSQL is a PostgreSQL-compatible relational database. You use familiar DDL commands to create new objects in the database and the psql \include or \copy meta-commands to load data.

You should have an open psql session connected to Aurora DSQL in us-east-1. If not, follow the instructions in the last section Connect to Aurora DSQL as admin to connect.

Run the following command in psql to create the database objects for our rewards points application. The script creates a schema called xpoints and all of the tables that our application will use.

```
\include ~/rewards-backend/src/main/sql/ddl.sql
```

The following is an excerpt of the contents of the ddl.sql script, showing the creation of the tables and indexes for our rewards points application. You don't need to run these statements now, since they were executed in the ddl.sql script above, but you can see how we've modeled our data for our application.

```
create schema xpoints;

create table xpoints.customers
(
    id          uuid primary key default gen_random_uuid(),
    username    varchar(50),
    first_name  varchar(50),
    last_name   varchar(50),
    maiden_name varchar(50),
    gender      varchar(10),
    email       varchar(50),
    phone_num   varchar(20),
    age         int,
    address     varchar(100),
    city        varchar(50),
    state       varchar(25),
    state_code  varchar(20),
    postal_code varchar(10)
);

create index on xpoints.customers (username);

create table xpoints.catalog_items
(
    id                  uuid primary key default gen_random_uuid(),
    name                varchar(50),
    description         varchar(200),
    category            varchar(20),
    usd_price           numeric(8,2),
    points_price        int,
    rating              real,
    sku                 varchar(20),
    weight              real,
    width               real,
    height              real,
    depth               real,
    thumbnail_id        uuid
);

create table xpoints.catalog_images
(
    item_id             uuid,
    image_id            uuid,
    primary key (item_id, image_id)
);

create index on xpoints.catalog_images (item_id);

create table xpoints.shopping_cart_items
(
    customer_id     uuid,
    item_id         uuid,
    quantity        int,
    primary key (customer_id, item_id)
);

create index on xpoints.shopping_cart_items (customer_id);

create table xpoints.points_balances
(
    customer_id     uuid primary key,
    points_balance  bigint not null
);

create table xpoints.transactions
(
    id                  uuid primary key default gen_random_uuid(),
    customer_id         uuid,
    tx_type             varchar(10),
    points              bigint,
    tx_dt               timestamp default now(),
    tx_description      varchar(50)
);

create index on xpoints.transactions (customer_id);

create table xpoints.order_items
(
    tx_id               uuid,
    cat_item_id         uuid,
    unit_cnt            int,
    unit_points_price   int,
    primary key (tx_id, cat_item_id)
);

create index on xpoints.order_items (tx_id);

create table xpoints.images
(
    id          uuid primary key default gen_random_uuid(),
    filename    varchar(100)
);

create table xpoints.image_urls
(
    image_id          uuid,
    region            varchar(20),
    presigned_url     varchar(2000),
    created           timestamp default now(),
    primary key (image_id, region)
```

```
);

create role rewards_ro with login;
grant usage on schema xpoints to rewards_ro;
grant select on all tables in schema xpoints to rewards_ro;

create role rewards_rw with login;
grant usage on schema xpoints to rewards_rw;
grant select, insert, update, delete on all tables in schema xpoints to
rewards_rw;
```

You can explore the schema and describe the objects in the database with psql commands, just as you would with any PostgreSQL database. For example, to list all tables in our xpoints schema:

```
\dt xpoints.*
```

To describe the xpoints.customers table, run the following:

```
\d xpoints.customers
```

Now load some sample data into our database using the load_rewards_data.sql script by running the following:

```
\i ~/rewards-backend/src/main/sql/load_rewards_data.sql
```

The contents of the load_rewards_data.sql script are shown below for reference. Each line uses \copy to load a Comma Separated Values (CSV) file containing rows of data for a single table. You do not need to run these instructions, since you already ran load_rewards_data.sql above.

```
\copy xpoints.images from '~/rewards-backend/src/main/sql/images.csv' csv
escape '\';
\copy xpoints.customers from '~/rewards-backend/src/main/sql/customers.csv'
csv escape '\';
\copy xpoints.points_balances from '~/rewards-backend/src/main/sql/
points_balances.csv' csv escape '\';
\copy xpoints.catalog_items from '~/rewards-backend/src/main/sql/
catalog_items.csv' csv escape '\';
\copy xpoints.catalog_images from '~/rewards-backend/src/main/sql/
catalog_images.csv' csv escape '\';
\copy xpoints.transactions from '~/rewards-backend/src/main/sql/
transactions.csv' csv escape '\';
\copy xpoints.order_items from '~/rewards-backend/src/main/sql/
order_items.csv' csv escape '\';
\copy xpoints.shopping_cart_items from '~/rewards-backend/src/main/sql/
shopping_cart_items.csv' csv escape '\';
```

Run the following query to see a few customer records:

```
select * from xpoints.customers limit 2;
```

Verify the second region

Switch to the browser tab or window for us-east-2. You should have an open psql session in us-east-2. If not, follow the instructions in Connect to Aurora DSQL as admin to open a psql session to Aurora DSQL in us-east-2.

You may see the following error message the first time you try to run a statement:

```
postgres=> begin;
ERROR:  schema has been updated by another transaction, please retry:
(OC001)
```

This error means the session had to read the updated information for the xpoints schema (all the new tables we created). Any subsequent transaction will run without any additional changes.

Run the following query to see that our data is also available in us-east-2.

```
select * from xpoints.customers limit 2;
```

```
postgres=> select * from xpoints.customers limit 2;
                 id                  |           username            | first_name | last_name |
maiden_name | gender |             email              | phone_n
um | age |    address     |   city    |      state      | state_code | postal_code
--------------------------------------+--------------------------------
+------------+-----------+------------+--------
+--------------------------------+--------
---+-----+---------------+----------+----------------+------------
+-------------
 0000a2c0-71c2-473a-856b-ffb89526d5e5 | carlos_salazar_121@example.mil
| Carlos     | Salazar   | García      | Other  | carlos_salazar_121@example.mil |
555-014
5  |  47 | 5425 O Street | Any Town | South Carolina | SC         | 42591
 0005c022-5872-4798-8d44-6fb8d7ef1a56 |
kwaku_mensah_108@example.net   | Kwaku      | Mensah    | Jayashankar | Male
| kwaku_mensah_108@example.net   | 555-011
6  |  24 | 8851 Q Street | Nowhere  | Virginia       | VA         | 94245
(2 rows)

postgres=>
```

Click Next to move on to the next section where we'll explore the data model in more depth.

Explore the data model

In this module we will explore the recommended data model and transactional patterns for Aurora DSQL databases.

Primary keys

Notice that all the tables we defined in module Create schema and load data have Primary Keys. This is of critical importance in Aurora DSQL. Primary Keys are not only used for data retrieval and better query performance, but also for records distribution. If you do not specify a Primary Key, Aurora DSQL will assign a hidden UUID value to each row. You cannot change or add a Primary Key to a table post-creation time. The main data model consideration you will make with Aurora DSQL is to choose a Primary Key for your larger or "hot" tables that provides reasonably random key distribution. Serial, monotonically incrementing integers as keys are an anti-pattern with Aurora DSQL.

Foreign keys

Aurora DSQL does not support foreign key constraints. Applications using Aurora DSQL should not rely on the database to enforce table dependencies. Instead, they can enforce their own order of operations. For example, deleting thousands of dependent rows for each parent row deletion will not need to happen in the same transaction. It can be postponed, broken down into smaller batches and processed during low traffic hours.

In a distributed system, you can use an event-driven architecture to maintain data consistency across different data stores, or to perform cascading operations asynchronously. For example, you can use AWS Lambda to maintain data consistency between a users table and a user_addresses table. When a user gets deleted, a Lambda function can be triggered to update the corresponding information in the user_addresses table.

The queries below are for demonstration. You don't need to execute them.

```
CREATE TABLE users (
  user_id INT PRIMARY KEY,
  name TEXT,
  email TEXT
);

CREATE TABLE user_addresses (
  user_id INT,
  address_id INT,
  primary key (user_id, address_id)
);

CREATE TABLE addresses (
  address_id INT PRIMARY KEY,
  address TEXT
);
```

In this scenario, your application might start by deleting the users record:

```
DELETE FROM users WHERE user_id = 123
```

Then the Lambda function would be triggered to perform the corresponding update operations:

```
DELETE FROM user_addresses WHERE user_id = 123
```

Use of indexes

In Aurora DSQL, indexes are created asynchronously. Let's create a secondary index on the phone_num column of the customers table in the xpoints schema to see asynchronous index creation in action. Run the following query:

```
create index async on xpoints.customers (phone_num);
```

You will see an output similar to the following:

```
postgres=> create index async on xpoints.customers (phone_num);
          job_id
----------------------------
 f3idlvjvendfnh3cxjjjozxssy
(1 row)
```

CREATE INDEX ASYNC statements do not block, and return immediately with a job_id. They do not hold any internal locks. You will be able to use the customers table while the index gets built. There is one aspect to pay attention to, though: the index creation will force a database catalog update, which needs to be propagated to all existing sessions (including the one having triggered the index creation). This means that after the index creation finishes, all active sessions in the database will receive a concurrency conflict on their next query and will need to retry their latest transaction:

```
ERROR:  schema has been updated by another transaction, please retry:
(OC001)
```

As you can see below, simply execute the statement again and it will complete. We'll demonstrate Aurora DSQL concurrency behaviors in the upcoming section Concurrency.

```
postgres=> create index async on xpoints.customers (phone_num);
          job_id
----------------------------
 tffb5x3bzvh2fnr4v7gmdp2kfe
```

```
(1 row)

postgres=> select * from xpoints.customers limit 2;
ERROR:  schema has been updated by another transaction, please retry:
(OC001)

postgres=> select * from xpoints.customers limit 2;
                 id                  |            username            | first_name | last_name |
maiden_name | gender |             email              | phone_n
um | age |    address     |   city   |     state      | state_code | postal_code
--------------------------------------+--------------------------------
+------------+-----------+------------+--------
+--------------------------------+--------
---+-----+---------------+----------+---------------+------------
+-------------
 0000a2c0-71c2-473a-856b-ffb89526d5e5 | carlos_salazar_121@example.mil
| Carlos     | Salazar   | García      | Other  | carlos_salazar_121@example.mil |
555-014
5  |  47 | 5425 O Street | Any Town | South Carolina | SC         | 42591
 0005c022-5872-4798-8d44-6fb8d7ef1a56 |
kwaku_mensah_108@example.net   | Kwaku      | Mensah    | Jayashankar | Male
| kwaku_mensah_108@example.net   | 555-011
6  |  24 | 8851 Q Street | Nowhere  | Virginia       | VA         | 94245
(2 rows)
```

Check the status of the index creation job in the sys.jobs view by running the query below.

```
select * from sys.jobs;
```

The status column will show a value of "completed" when the index is ready.

```
postgres=> select * from sys.jobs;
          job_id           |  status   | details
----------------------------+-----------+---------
 tffb5x3bzvh2fnr4v7gmdp2kfe | completed |
(2 rows)
```

You can also check the status of the index in the pg_index system view. Run the following query:

```
select indisvalid from pg_index where indexrelid
='xpoints.customers_phone_num_idx'::regclass::oid;
```

The indisvalid column in pg_index is a boolean that indicates if

the index is currently valid for queries

. A value of t for indisvalid means that our index is ready to use. Run the query below to get the status of our index.

```
postgres=> select indisvalid from pg_index where indexrelid
='xpoints.customers_phone_num_idx'::regclass::oid;
 indisvalid
------------
 t
(1 row)
```

Now run the statement below to describe the table. You'll see our index customers_phone_num_index in the table description.

```
\d xpoints.customers
```

```
postgres=> \d xpoints.customers
                    Table "xpoints.customers"
   Column    |          Type          | Collation | Nullable |      Default
-------------+------------------------+-----------+----------
+-------------------
 id          | uuid                   |           | not null | gen_random_uuid()
 username    | character varying(50)  |           |          |
 first_name  | character varying(50)  |           |          |
 last_name   | character varying(50)  |           |          |
 maiden_name | character varying(50)  |           |          |
 gender      | character varying(10)  |           |          |
 email       | character varying(50)  |           |          |
 phone_num   | character varying(20)  |           |          |
 age         | integer                |           |          |
 address     | character varying(100) |           |          |
 city        | character varying(50)  |           |          |
 state       | character varying(25)  |           |          |
 state_code  | character varying(20)  |           |          |
 postal_code | character varying(10)  |           |          |
Indexes:
    "customers_pkey" PRIMARY KEY, remote_btree_index (id) INCLUDE
(username, first_name, last_name, maiden_name, gender, email, phone_num,
age, address, city, state, state_code, postal_code)
    "customers_phone_num_idx" remote_btree_index (phone_num)
    "customers_username_idx" remote_btree_index (username)
```

You can see the new index being used by queries that use the phone_num column:

```
explain select phone_num from xpoints.customers limit 1;
```

```
postgres=> explain select phone_num from xpoints.customers limit 1;
                                      QUERY PLAN
```

```
--------------------------------------------------------------------------------
-------------------------------
 Limit  (cost=725.03..733.04 rows=1 width=58)
   ->  Index Only Scan using customers_phone_num_idx on customers
(cost=725.03..40755.03 rows=5000 width=58)
        Projected via pushdown compute engine: phone_num
(3 rows)
```

JOIN, GROUP BY, ORDER BY, and Aggregate

Aurora DSQL supports the familiar relational operations: sorting, grouping, and aggregation. For example, the query below tells us what the most popular items are in our our catalog by counting the number of units sold for each item and sorting the results in descending order by the number of units sold. Catalog items with no sales are omitted from the list.

```
SELECT
  name, sum(unit_cnt) AS sum_items
FROM
  xpoints.catalog_items
  LEFT JOIN xpoints.order_items ON xpoints.catalog_items.id =
xpoints.order_items.cat_item_id
GROUP BY name
HAVING sum(unit_cnt) > 0
ORDER BY sum_items DESC, name;
```

Explain Plans

Aurora DSQL's EXPLAIN plan output will resemble PostgreSQL's.

```
EXPLAIN SELECT * FROM xpoints.customers limit 1;
\d+ xpoints.customers;

postgres=> EXPLAIN SELECT * FROM xpoints.customers limit 1;
                                                    QUERY PLAN

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
---

 Limit  (cost=725.03..733.04 rows=1 width=1206)
   ->  Index Only Scan using customers_pkey on customers
(cost=725.03..40755.03 rows=5000 width=1206)
        Projected via pushdown compute engine: id, username, first_name,
last_name, maiden_name, gender, email, phone_num, age, address, city, state,
state_code
, postal_code
(3 rows)
postgres=> \d+ xpoints.customers;
                                 Table "xpoints.customers"
```

```
  Column    |          Type          | Collation | Nullable |      Default      | Storage  |
Compression | Stats target | Description
-------------+------------------------+-----------+----------
+-------------------+----------+-------------+--------------+-------------
 id          | uuid                   |           | not null | gen_random_uuid() | plain    |
|             |
 username    | character varying(50)  |           |          |                   | extended |
|             |
 first_name  | character varying(50)  |           |          |                   | extended |
|             |
 last_name   | character varying(50)  |           |          |                   | extended |
|             |
 maiden_name | character varying(50)  |           |          |                   | extended |
|             |
 gender      | character varying(10)  |           |          |                   | extended |
|             |
 email       | character varying(50)  |           |          |                   | extended |             |
|
 phone_num   | character varying(20)  |           |          |                   | extended |
|             |
 age         | integer                |           |          |                   | plain    |             |             |
 address     | character varying(100) |           |          |                   | extended |
|             |
 city        | character varying(50)  |           |          |                   | extended |             |
|
 state       | character varying(25)  |           |          |                   | extended |             |
|
 state_code  | character varying(20)  |           |          |                   | extended |
|             |
 postal_code | character varying(10)  |           |          |                   | extended |
|             |
Indexes:
    "customers_pkey" PRIMARY KEY, remote_btree_index (id) INCLUDE
(username, first_name, last_name, maiden_name, gender, email, phone_num,
age, address, city, state, state_code, postal_code)
    "customers_phone_num_idx" remote_btree_index (phone_num)
    "customers_username_idx" remote_btree_index (username)
Access method: remote_btree_table
```

You might notice that the access path uses an index that includes all of the columns in the table. This is not an index that we have created. It is an index that the Aurora DSQL storage nodes use internally to access data. Aurora DSQL tables are index-organized tables, and not heap tables.

If we select from an indexed column, or when the predicate can be written as an equality, the explain plan will show that a

pushdown compute engine is used:

```
EXPLAIN SELECT phone_num FROM xpoints.customers limit 1;

postgres=> EXPLAIN SELECT phone_num FROM xpoints.customers limit 1;
                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Limit  (cost=725.03..733.04 rows=1 width=58)
   ->  Index Only Scan using customers_phone_num_idx on customers
(cost=725.03..40755.03 rows=5000 width=58)
         Projected via pushdown compute engine: phone_num
(3 rows)
```

Pushdown is an optimization technique that moves data processing closer to the data source (in Aurora DSQL's case, to the storage nodes). For example, imagine a query to a database filtered by a WHERE clause. If no pushdown happens, all the data will have to read into memory, and then the rows that do not apply to the filter will need to be removed. By contrast, if pushdown happens, the WHERE clause will be applied directly by the storage nodes, and only the rows that apply to the filter will be sent back to the query processors. Less data to be transferred, less work for the query processors, better performance!

Click Next to continue.

Concurrency Control

In any database that supports concurrent access, there is the risk of collision. Multiple transactions may try to update the same data at the same time. A good data model will minimize opportunities for collision, but collisions are always possible and must be managed. Databases provide ways of managing concurrent access to data. The two primary approaches are pessimistic concurrency control and optimistic concurrency control.

The PostgreSQL open-source database engine uses pessimistic concurrency control. With this approach, you lock a record before updating it. If another transaction is holding a lock on the record, you must wait for the lock to be released before proceeding. You must decide how long to wait on the lock before giving up. If you give up, you must decide what to do about your transaction. Will you report an error or will you re-attempt the transaction?

Aurora DSQL uses *optimistic* concurrency control, which is more common in non-relational databases than relational databases. With optimistic concurrency, you execute your transaction logic with little consideration for other transactions that may be trying to update the same data. When you've done your work, you attempt to commit the transaction. Aurora DSQL checks to see if writes in other concurrent transactions have interfered with writes in your transaction. If not, your transaction successfully commits. Otherwise, the

database reports an error to your transaction. You must then decide what to do, just as before. For most use cases, re-trying the transaction is the best approach.
With Aurora DSQL optimistic concurrency control, the first transaction to commit wins. Other colliding transactions fail with an error when they try to commit, even if they were *started* first.
In this section, we'll demonstrate Aurora DSQL optimistic concurrency behaviors by creating conflicting transactions from psql running in multiple terminal sessions.

Setup

You should have a browser tab or window opened to the AWS Console in the **us-east-1** region and one in the **us-east-2** region. Each tab or window should have a CloudShell terminal session open.
Login to the Aurora DSQL cluster from a CloudShell terminal session in each region. The Connect to Aurora DSQL as admin page provides instructions for connecting to Aurora DSQL with psql. However, the workshop environment has a convenience script to make it a little easier. In each region, run the command below, replacing <<YOUR CLUSTER ENDPOINT>> with the endpoint for the Aurora DSQL cluster in that region.

```
psqli.sh <<YOUR CLUSTER ENDPOINT>>
```

You should now have open psql sessions in both **us-east-1** and **us-east-2**.

Handling CloudShell Timeouts

CloudShell sessions end after 20-30 minutes of inactivity, which may occur while following the steps in this workshop. If you see a timeout message or if commands produce error messages or don't seem to run as expected, re-initialize the shell session by running the following commands:

```
./setup.sh
. ./.bashrc
```

Read-write transactions

One of our customers, Carlos Salazar, has moved. We must change his address. Run the following SQL statements in the us-east-1 psql session.

Run all statements

In this section, we induce concurrency conflicts by opening a transaction, updating a row *without committing* and then committing a change to the same row in another region before trying to commit in the first region. It's important that you copy all of the statements in the examples for this to work. Use the "Copy content" button to the right of the code boxes to copy the statements correctly.

```
begin;
select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
update xpoints.customers set address = '123 Main Street' where username =
'carlos_salazar_125@example.edu';
```

Notice that we have not committed our transaction yet. Your output should look like this so far:

```
postgres=> begin;
postgres=*> select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
    address
----------------
 4550 Z Street
(1 row)

postgres=*> update xpoints.customers set address = '123 Main Street' where
username = 'carlos_salazar_125@example.edu';
postgres=*>
```

Now switch to the psql session in us-east-2. Run the following query statements:

```
begin;
select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
```

Our output should look like this:

```
postgres=> begin;
postgres=*> select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
    address
----------------
 4550 Z Street
(1 row)

postgres=*>
```

Notice that we still see the original address value since our other transaction has not committed yet. Now run the following SQL statements, still in us-east-2:

```
update xpoints.customers set address = '201 Rocky Blvd' where username =
'carlos_salazar_125@example.edu';
commit;
```

Our transaction is now committed. Switch back to the us-east-1 and run the following statement:

```
commit;
```

You should see the following error:

```
postgres=*> commit;
ERROR:  change conflicts with another transaction, please retry: (OC000)
postgres=>
```

The first transaction to commit the update succeeded. However, when the second concurrent transaction attempted to commit, Aurora DSQL recognized that Carlos Salazar's record had been modified by the first transaction and returned an error, even though the transactions occurred in separate regions! Notice that neither transaction attempted to lock the row before attempting to update it.

Run the following query in us-east-1 to verify which transaction succeeded:

```
select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
```

Your output should look like this:

```
postgres=> select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
   address
-----------------
 201 Rocky Blvd
(1 row)
```

Note that concurrency protection only applies to concurrent transactions. In the us-east-1 psql session, execute the following transaction:

```
begin;
update xpoints.customers set address = '221B Baker Street' where username =
'carlos_salazar_125@example.edu';
commit;
```

Switch to the us-east-2 psql session and execute the following transaction:

```
begin;
update xpoints.customers set address = '999 Wrong Way' where username =
'carlos_salazar_125@example.edu';
commit;
```

Now run the following query:

```
select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
```

Your output should look like this:

```
postgres=> select address from xpoints.customers where username =
'carlos_salazar_125@example.edu';
   address
---------------
 999 Wrong Way
(1 row)
```

Note that both transactions succeeded and we have the latest address from the second query. The two transactions did not conflict with one another because the second transaction began after the first transaction had already committed.
Click Next to continue.

Read-only transactions

Let's continue exploring concurrency in Aurora DSQL. Execute the following statements in the psql session in **us-east-1**:

Run all statements
It's important that you copy all of the statements in the examples for them to work. Use the "Copy content" button to the right of the code boxes to copy the statements correctly.

```
begin;
select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
```

Note Mr. Salazar's age in our output:

```
postgres=> begin;
postgres=*> select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
 age
-----
  65
(1 row)
```

Switch to psql session in **us-east-2** and run the following statement:

```
update xpoints.customers set age = 99 where username =
'carlos_salazar_125@example.edu';
```

Now switch back to the **us-east-1** session and run the following statements:

```
select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
commit;
```

Our output looks like this:

```
postgres=> begin;
postgres=*> select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
 age
-----
  65
(1 row)

postgres=*> select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
```

```
 age
-----
  65
(1 row)

postgres=*> commit;
postgres=>
```

Notice that when we ran the second SELECT statement, we received the same result for Mr. Salazar's age, 65, not the updated value from the other transaction. This is because Aurora DSQL uses snapshot isolation, which requires *repeatable reads*.

Also note that our transaction successfully committed, even though it read a row that was updated in a concurrent transaction. Aurora DSQL does not perform concurrency checks for transactions that do not update data. This complies with snapshot isolation.

Now run the query again in the **us-east-1**:

```
select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
```

Since this is a new transaction, we'll see the latest value:

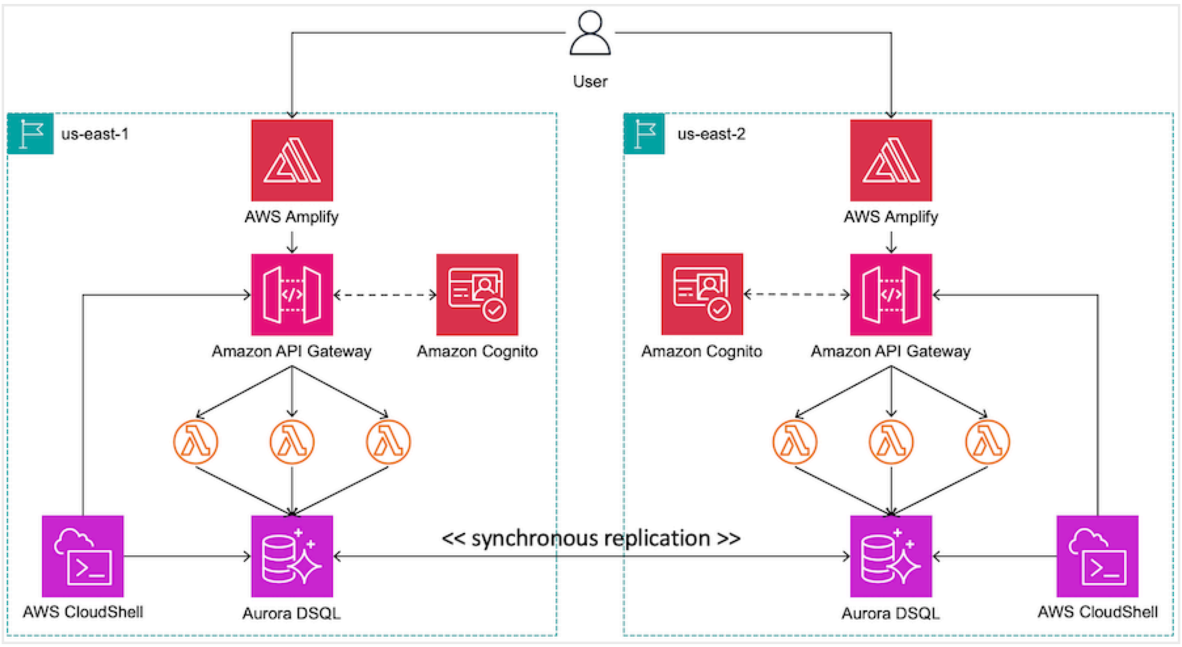
```
postgres=> select age from xpoints.customers where username =
'carlos_salazar_125@example.edu';
 age
-----
  99
(1 row)

postgres=>
```

Click **Next** to continue.

3. Building the Rewards App

Now that we've explored Aurora DSQL's features using psql and learned how Aurora DSQL optimistic concurrency works, let's deploy a multi-region active-active web application that takes advantage of Aurora DSQL's bi-directional synchronous replication to accept reads and writes in two AWS regions.

The application is a retail rewards points system that allows customers to browse a product catalog and redeem loyalty points for merchandise from the catalog. The application consists of a set of AWS Lambda functions behind an API Gateway with a web interface hosted in AWS Amplify. The application uses a Cognito user pool for authentication and IAM for authorization. The web interface interacts with the application through the API hosted by API Gateway. The application is not aware that it is multi-region. It simply interacts with its local Aurora DSQL database endpoint, letting Aurora DSQL manage data replication and strong consistency across the regions.



You should have a browser tab or window opened to the AWS Console in the **us-east-1** region and one in the **us-east-2** region. Each tab or window should have a CloudShell session open. You'll need an open psql session to Aurora DSQL in either of the regions.
Click **Next** to continue.

Deploy the application

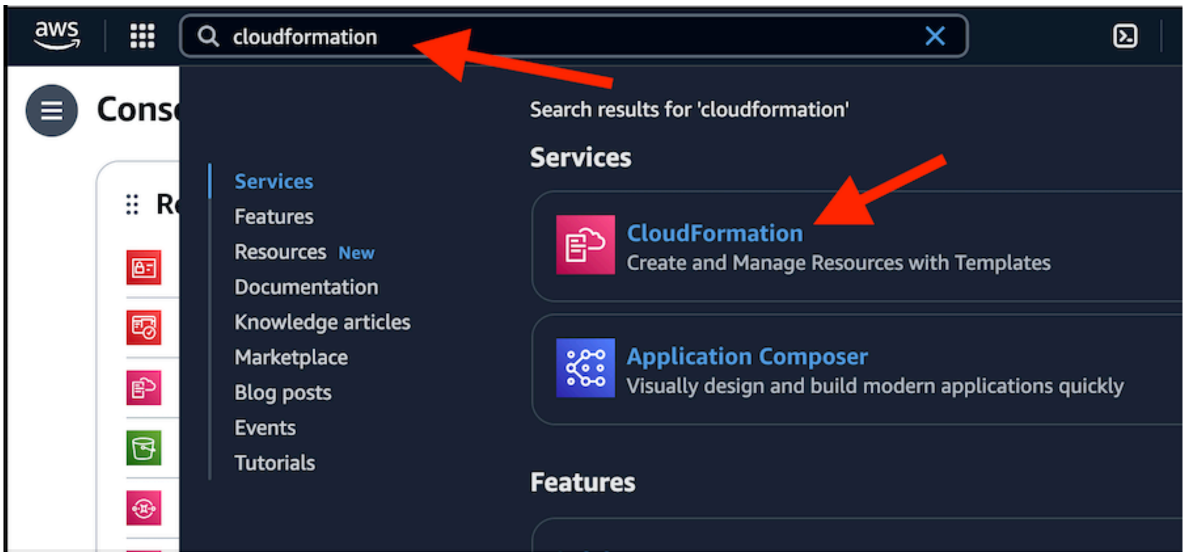
In this section, we'll launch the Rewards Points application in **us-east-1**. You'll need the following information:

Parameter	Description
Template S3 URL	This is the S3 path of the CloudFormation template to launch the app. It was output as "App Template URL" by the setup.sh script that you ran earlier in CloudShell. You can also get the value from outputs.txt in the CloudShell environment. Run cat outputs.txt in CloudShell to display the file's contents.
CodeBucketName	This is the name of the S3 bucket that was created by the setup.sh script. It was output as "Code Bucket". You can also get the value from outputs.txt in the CloudShell environment.

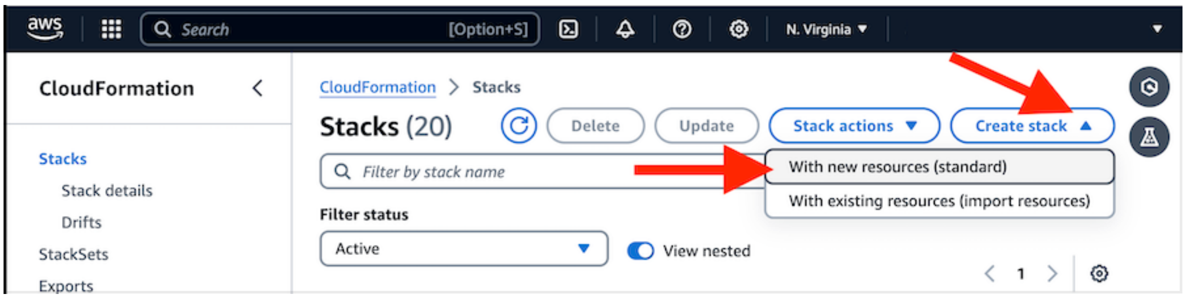
ClusterEndpoint	Get the endpoint of the Aurora DSQL cluster from the Aurora DSQL console for the region you're deploying to. Note that this is the cluster's full endpoint, not its ID. This endpoint is specific to the region you're deploying in. Each region will have its own Aurora DSQL cluster endpoint.

Deploy the application back-end

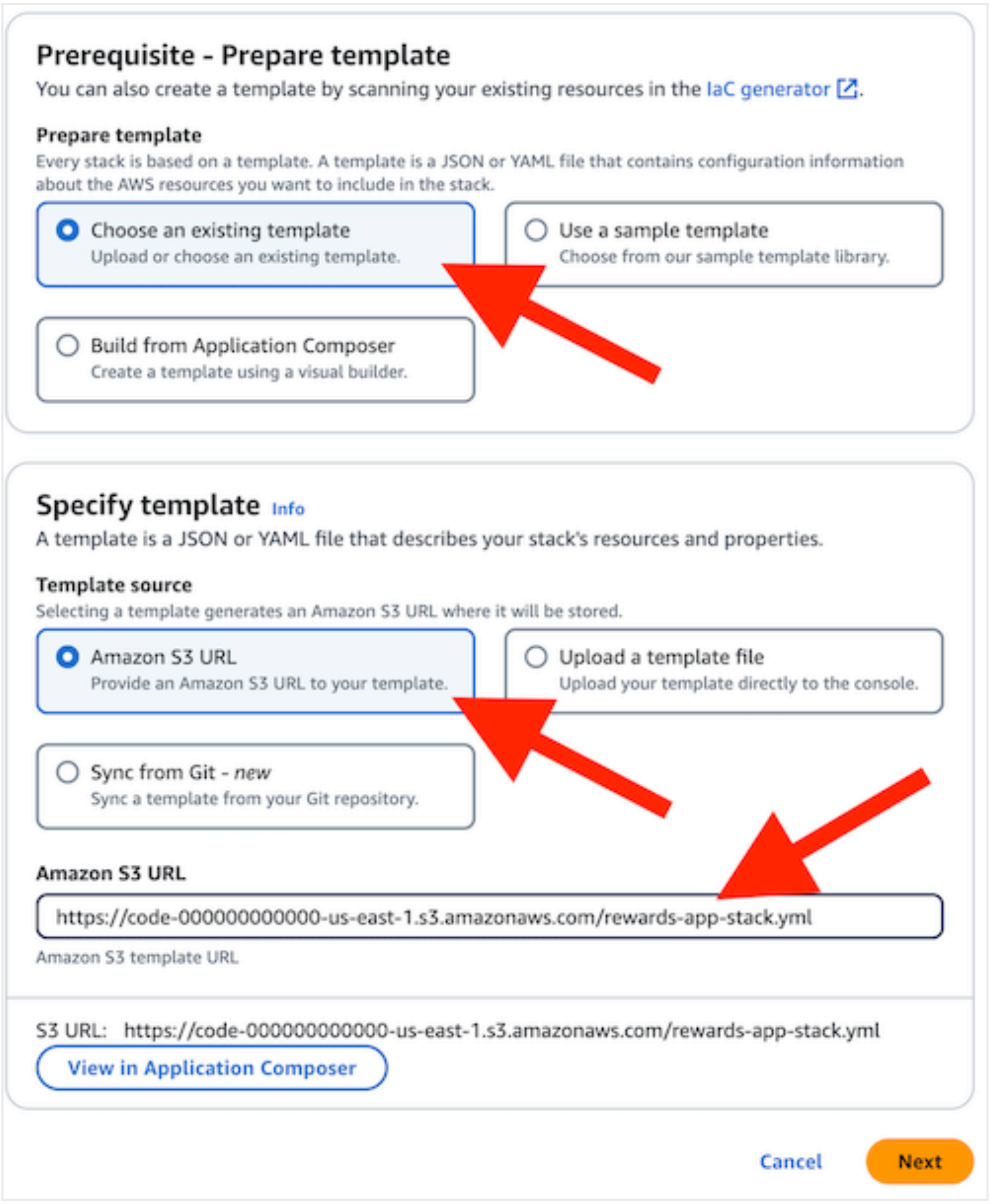
In the browser tab for **us-east-1**, navigate to CloudFormation by typing "CloudFormation" into the search bar at the top of the AWS Console page and clicking the link for CloudFormation.



Click the **Create stack** button and select "With new resources (standard)".



On the next page, leave the **Choose an existing template** and **Amazon S3 URL** radio buttons selected. In the **Amazon S3 URL** field, enter the value of **Template S3 URL** from the table above. Then click **Next**.


Prerequisite - Prepare template

You can also create a template by scanning your existing resources in the IaC generator.

Prepare template

Every stack is based on a template. A template is a JSON or YAML file that contains configuration information about the AWS resources you want to include in the stack.

[✓ Choose an existing template — Upload or choose an existing template.]
Use a sample template — Choose from our sample template library.
Build from Application Composer — Create a template using a visual builder.

Specify template

A template is a JSON or YAML file that describes your stack's resources and properties.

Template source

Selecting a template generates an Amazon S3 URL where it will be stored.

[✓ Amazon S3 URL — Provide an Amazon S3 URL to your template.]
Upload a template file — Upload your template directly to the console.
Sync from Git - *new* — Sync a template from your Git repository.

Amazon S3 URL

https://code-000000000000-us-east-1.s3.amazonaws.com/rewards-app-stack.yml

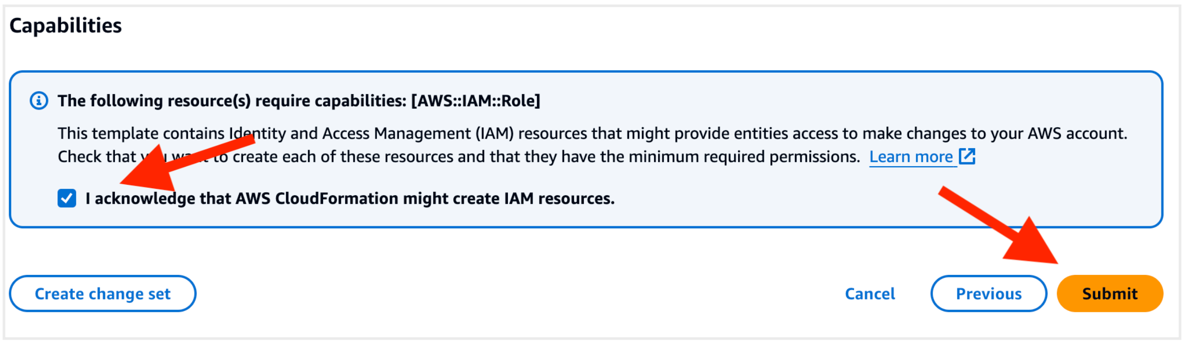
Amazon S3 template URL

S3 URL: https://code-000000000000-us-east-1.s3.amazonaws.com/rewards-app-stack.yml

On the next page, enter "aurora-dsql-rewards" for the **Stack name**. Be sure to enter the stack name exactly as shown or subsequent steps may not work. In the **Parameters** section, populate the **CodeBucketName** and **ClusterEndpoint** from the table above. Leave the remaining parameters at their default values.

Enter the stack name EXACTLY as shown or the web front-end will not work! Now scroll to the bottom of the **Configure stack options** page and click **Next**. Scroll to the bottom of the **Review and create** page and click the **I acknowledge that AWS CloudFormation might create IAM**

resources. prompt and click **Submit**.



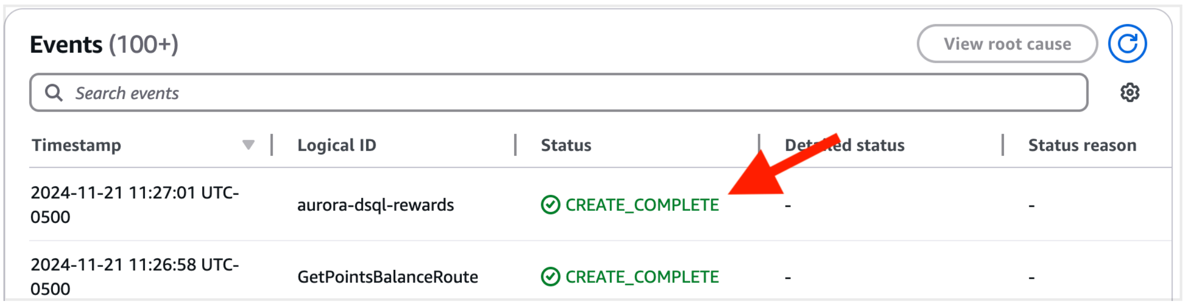
Capabilities

The following resource(s) require capabilities: [AWS::IAM::Role]

This template contains Identity and Access Management (IAM) resources that might provide entities access to make changes to your AWS account. Check that you want to create each of these resources and that they have the minimum required permissions. Learn more

☑ **I acknowledge that AWS CloudFormation might create IAM resources.**

After several minutes, the stack will be deployed. In the **Events** tab in CloudFormation, you should see a green "CREATE_COMPLETE" message, indicating successful deployment.

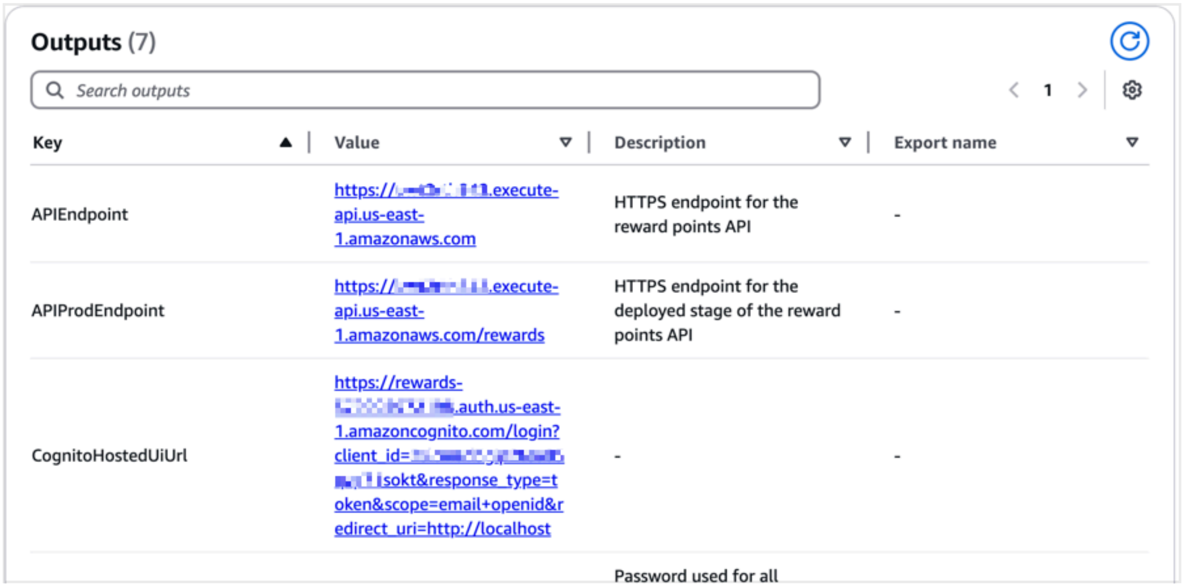


Events (100+)

Timestamp	Logical ID	Status	Detailed status	Status reason
2024-11-21 11:27:01 UTC-0500	aurora-dsql-rewards	✓ CREATE_COMPLETE	-	-
2024-11-21 11:26:58 UTC-0500	GetPointsBalanceRoute	✓ CREATE_COMPLETE	-	-

Click on the **Outputs** tab to see the stack outputs. These include information that we'll need in upcoming sections, including the application's API endpoint in **us-east-1** and information for logging-in to Amazon Cognito

.



Outputs (7)

Key	Value	Description	Export name
APIEndpoint	https://[redacted].execute-api.us-east-1.amazonaws.com	HTTPS endpoint for the reward points API	-
APIProdEndpoint	https://[redacted].execute-api.us-east-1.amazonaws.com/rewards	HTTPS endpoint for the deployed stage of the reward points API	-
CognitoHostedUiUrl	https://rewards-[redacted].auth.us-east-1.amazoncognito.com/login?client_id=[redacted]isokt&response_type=token&scope=email+openid&redirect_uri=http://localhost	-	-
		Password used for all	

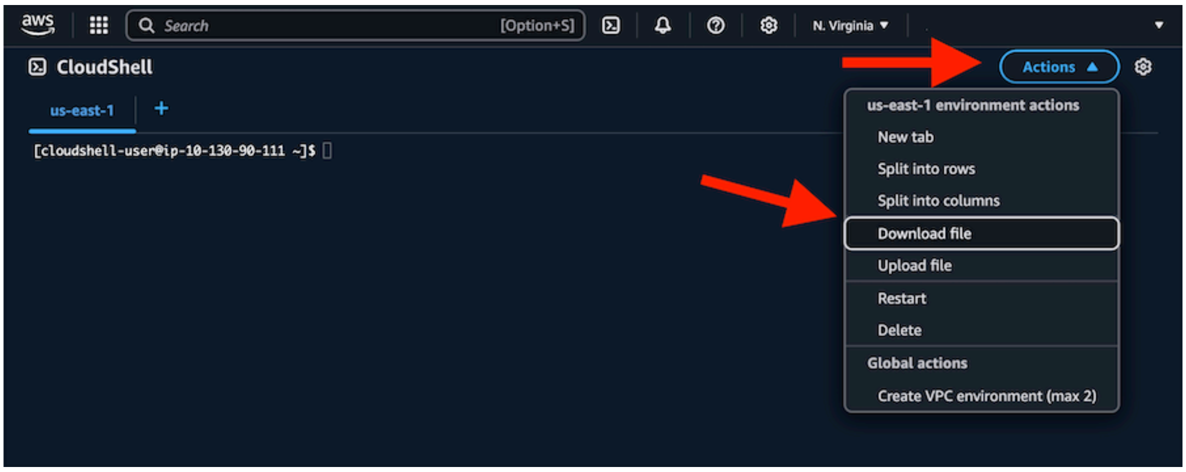
Deploy the application web interface

Now that the back-end APIs are deployed, let's deploy the web application to AWS Amplify in **us-east-1**.

Go to the CloudShell terminal session for **us-east-1**. Run the command below to configure the web application deployment ZIP file. Note that the "dot" at the beginning of the command is not a mistake. This allows the script to set some local environment variables.

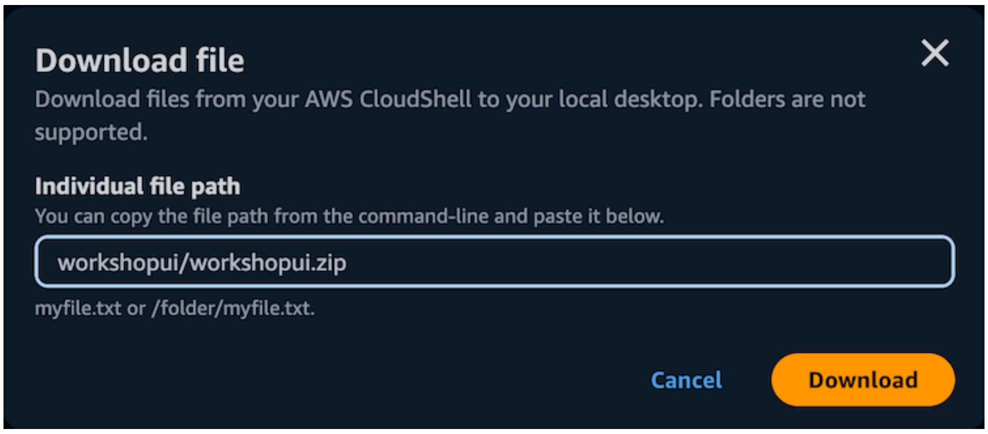
```
. config_ui.sh
```

Download the workshop user interface ZIP file from CloudShell by clicking the **Actions** drop-down menu and then **Download file**.



```
[cloudshell-user@ip-10-130-90-111 ~]$
```

At the **Individual file path** prompt, enter "workshopui/workshopui.zip".



Download file

Download files from your AWS CloudShell to your local desktop. Folders are not supported.

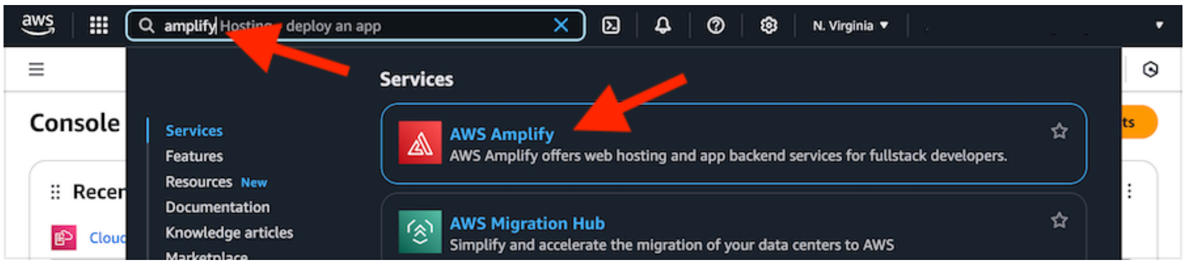
Individual file path

You can copy the file path from the command-line and paste it below.

workshopui/workshopui.zip

myfile.txt or /folder/myfile.txt.

In the browser tab for **us-east-1**, navigate to AWS Amplify by typing "Amplify" into the search bar at the top of the AWS Console page and clicking the link for AWS Amplify.



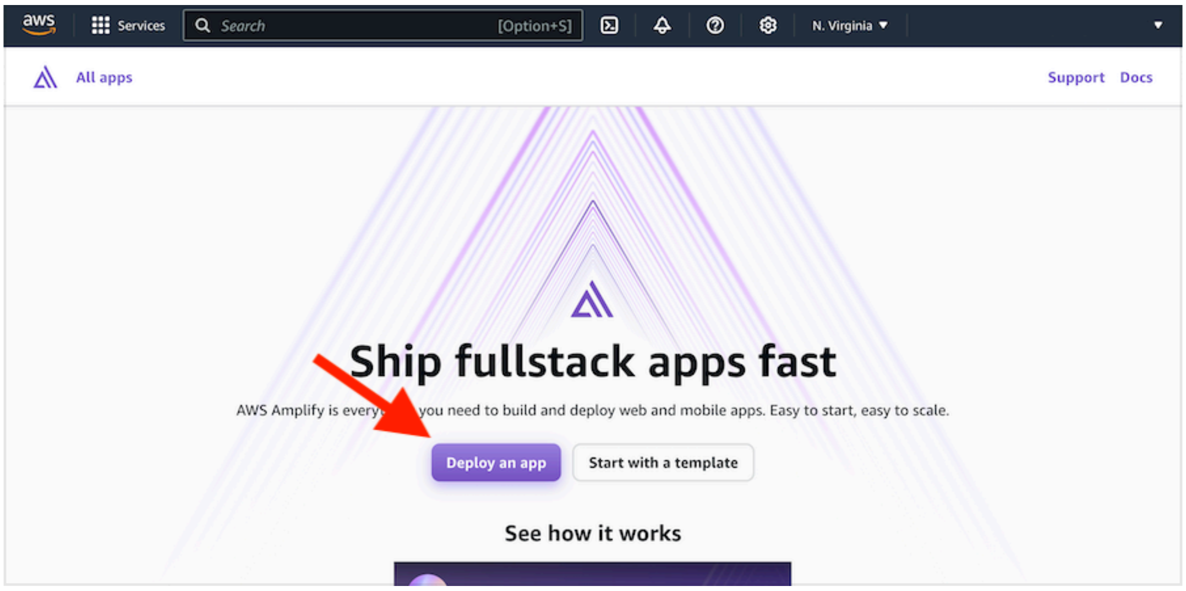
AWS Amplify

AWS Amplify offers web hosting and app backend services for fullstack developers.

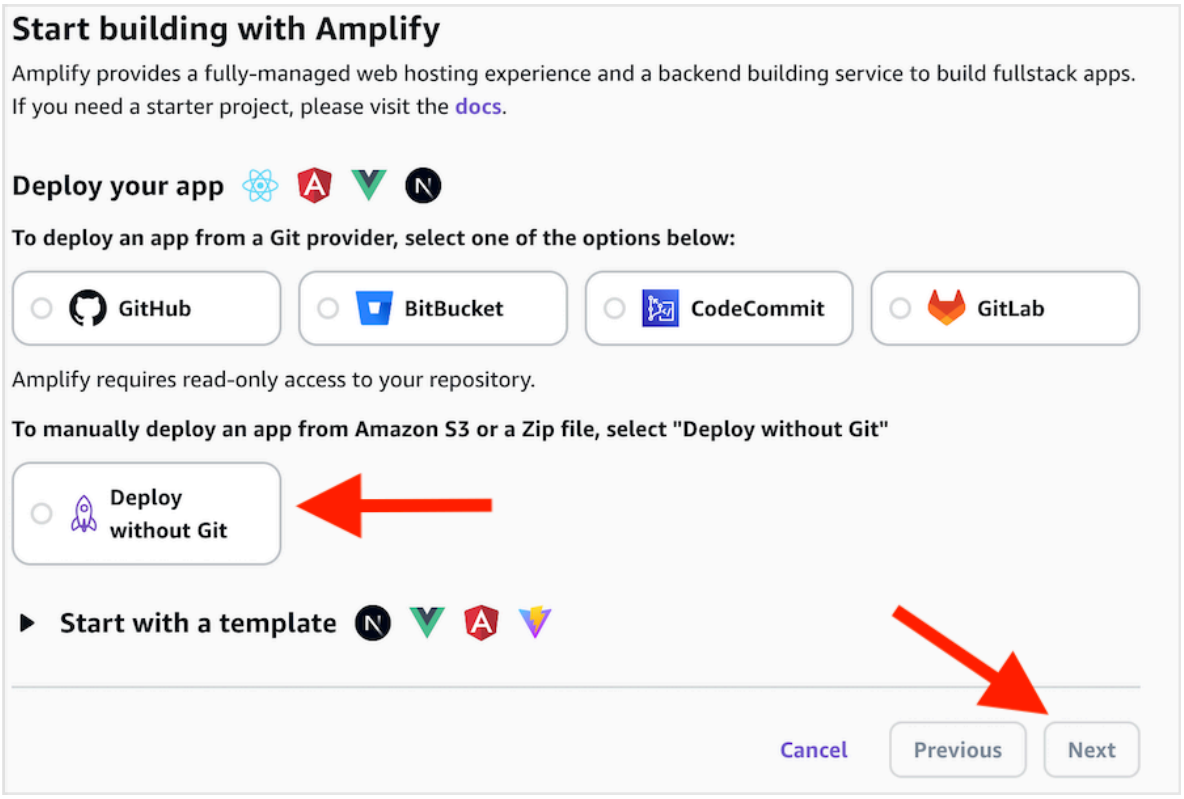
AWS Migration Hub

Simplify and accelerate the migration of your data centers to AWS

In the Amplify welcome page, click **Deploy an app**.



Click **Deploy without Git** and click **Next**.



Start building with Amplify

Amplify provides a fully-managed web hosting experience and a backend building service to build fullstack apps. If you need a starter project, please visit the docs.

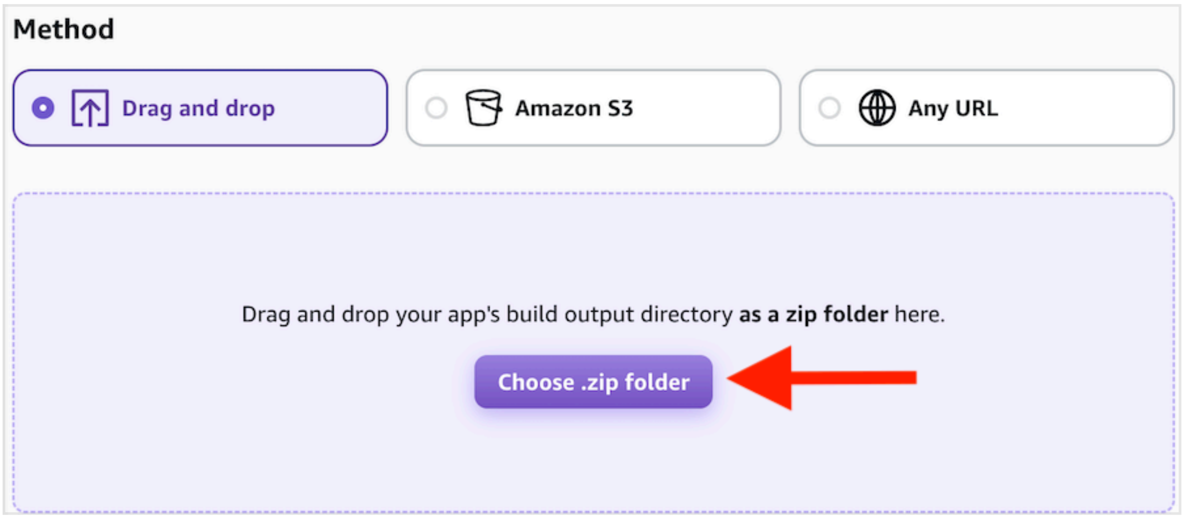
Deploy your app

To deploy an app from a Git provider, select one of the options below:

Amplify requires read-only access to your repository.

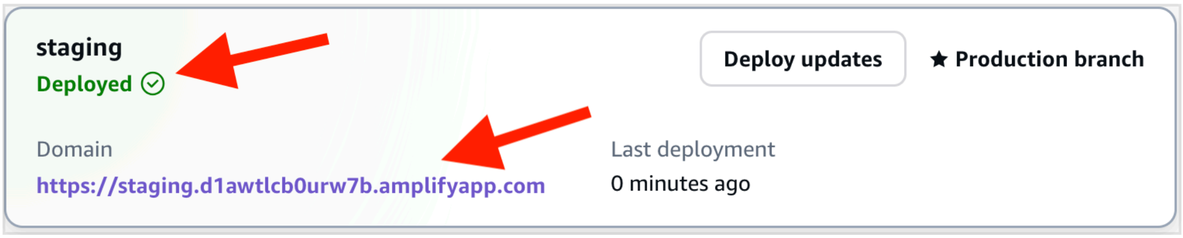
To manually deploy an app from Amazon S3 or a Zip file, select "Deploy without Git"

Drag and drop the downloaded workshopui.zip file into the purple drag and drop area, then click **Save and deploy**.



Drag and drop your app's build output directory **as a zip folder** here.

When Amplify finishes deploying your application, it will show you a "Deployed" status and the URL for your application.



Domain
https://staging.d1awtlcb0urw7b.amplifyapp.com

Last deployment
0 minutes ago

We'll begin using the UI in a moment, so make note of the application's URL for this region. On your local workstation, delete the workshopui.zip file that you downloaded from the CloudShell terminal.

Deploy the application in us-east-2

Now we'll deploy the application in **us-east-2**. Switch over to your browser tab for **us-east-2** and repeat all of the steps above, replacing the values of **Template S3 URL**, **CodeBucketName**, and **ClusterEndpoint** with the correct values for **us-east-2**. They will not be the same as the values for **us-east-1**! When the CloudFormation stack finishes deploying, click on the **Outputs** tab to see the stack's output values for **us-east-2**. Complete the steps to configure and deploy the application in **us-east-2**. Make note of the application's URL from Amplify. Finally, delete the workshopui.zip file from your workstation.
In the next lesson, we'll begin using our application in both regions to see an active-active application in action! Click **Next** to continue.

Explore application

Now that we've deployed our rewards points application to both regions, let's use it to shop for items in multiple regions to see Aurora DSQL active-active transactions in action.

Get login information

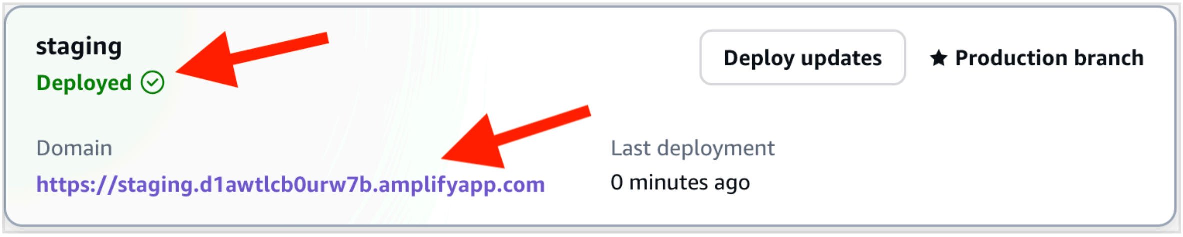
We'll need to login to the application, so let's find a user to login as. If you don't already have an open Aurora DSQL session in **us-east-1**, connect to Aurora

DSQL following the instructions found in Connect to Aurora DSQL. Select a customer's username from the xpoints.customers table. You can run the following query to fetch a sample of customers and simply pick one from the list.
select username from xpoints.customers ORDER BY random() limit 20;

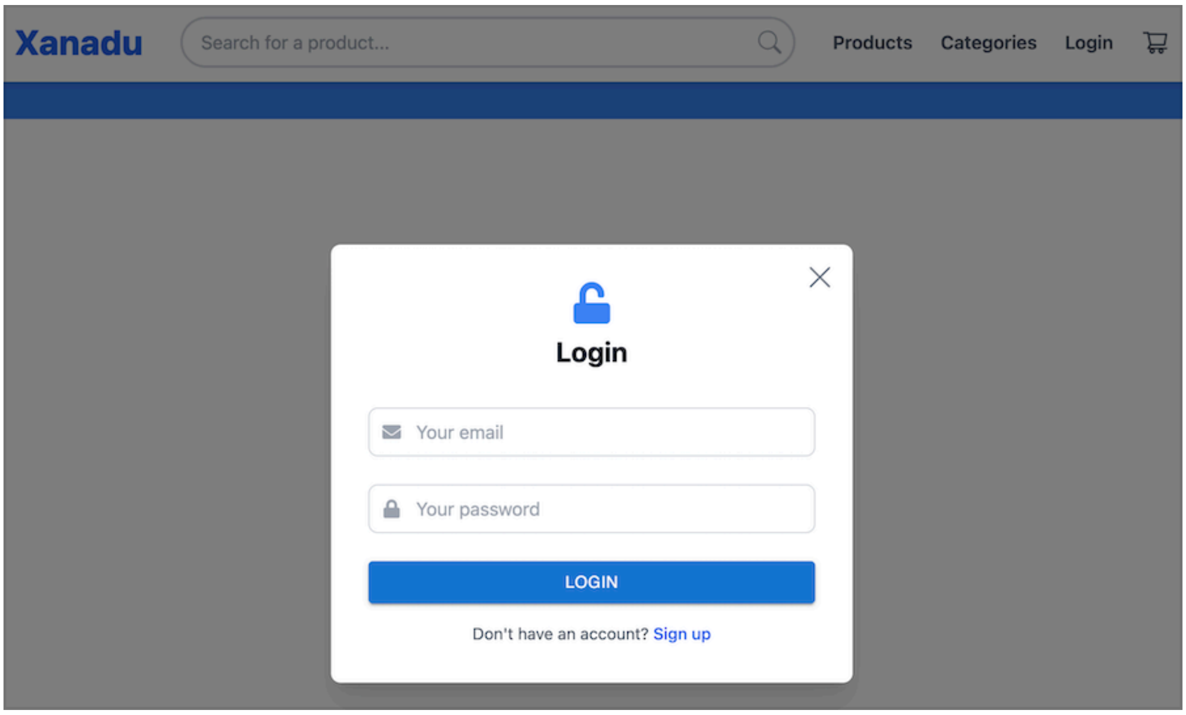
The users from xpoints.customers are loaded into Cognito asynchronously. There are several thousand customer records in our database, so it may take several minutes for all of them to load into Cognito. If you can't login with the customer you've selected, try another. You can retry your customer again after several minutes.
Next you'll need the user's password. All users were given the same password in Cognito. You can find it in the CloudFormation stack's outputs under the name CognitoUserPassword or you can find it in your CloudShell session in either region by running the command:
echo $REWARDS_PASSWORD

Login to the app in both regions

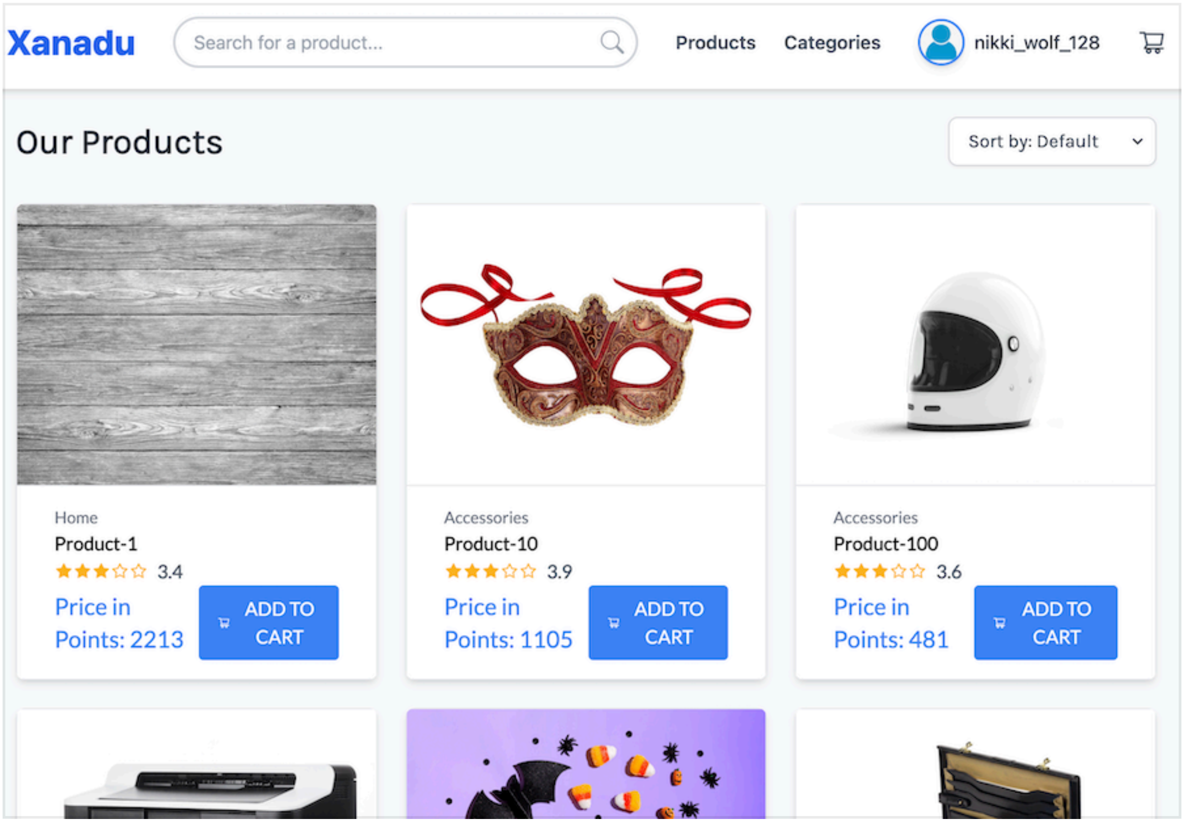
In this section, we'll use the rewards points applications in two regions. Open the rewards points application in **us-east-1** by pasting its link from Amplify into a new browser tab or by clicking on the application's deploy link in Amplify.


staging
Deployed
Domain
https://staging.d1awtlcb0urw7b.amplifyapp.com
Last deployment
0 minutes ago

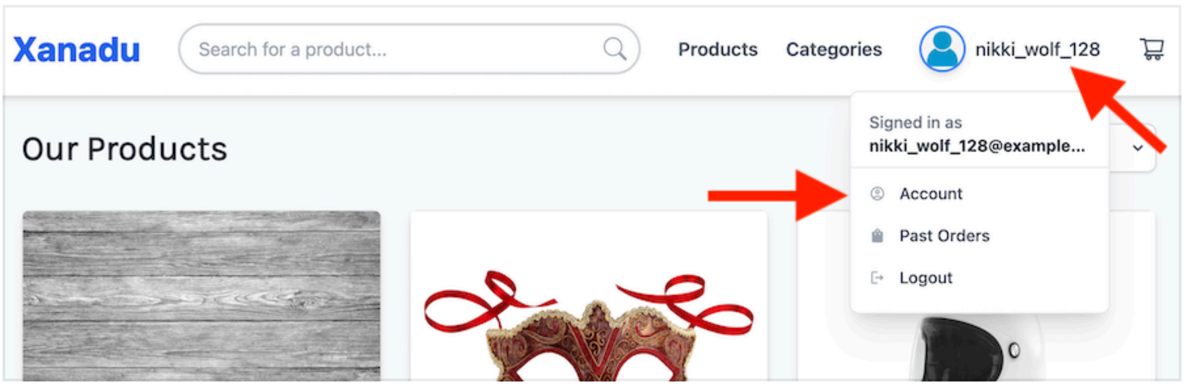
You'll be presented with a login page.



Enter the username you chose from the xpoints.customers table and the password from above and click **Login**. You'll see the application's home page presenting the catalog of all of the great merchandise you can redeem with your rewards points.



Spend a few moments exploring the application. Click on a product to see its product details. Click the user profile icon in the top-right corner of the app and click **Account** to see this user's points balance. Click the user profile icon again and click **Past Orders** to see if your user has any purchase history.



Now do the same for **us-east-2**, clicking the application link in Amplify for **us-east-2** and logging in again as the **same** user (multi-region single sign-on is out of scope for this workshop).
You should now have two browser tabs or windows logged into the rewards points application, one session in each region. Each session communicates with the API and back-end for its region, which connects to the Aurora DSQL endpoint for that region. In the next section, we'll flip back and forth between the sessions to make updates and view them in the other region.

Active-active transactions

In the rewards points app in **us-east-1**, find a product you like and click **Add to Cart**. Click the shopping cart logo in the upper-right corner to see the contents of your cart.
Now switch to the rewards points app in **us-east-2** and refresh your browser page. The shopping cart icon in the upper-right corner should now indicate that there's an item in the cart. Click the shopping cart item. The item in the cart should be the item you selected in **us-east-1**.
Click **Products** in the main navigation (still in **us-east-2**). Find another product that you like and add it to the cart. Click the cart icon to view the contents of your cart.
Now switch back to the rewards points app in **us-east-1** and refresh your browser page. The shopping cart icon now shows that there are two items in your shopping cart. Click the shopping cart icon to view the cart's contents. As you can see, cross-region replication in Aurora DSQL is bi-directional. Both regions are always consistent.
Now click **Proceed to Checkout** and **Confirm Order**, still in **us-east-1**. Then click the user profile icon and select **Past Orders**. You should see your purchase transaction listed.
Switch back to the rewards points app in **us-east-2**. Click the user profile icon and select **Past Orders**. You should see the same list of transactions as you saw in **us-east-1**.
Congratulations! You have built and deployed a multi-region active-active Amazon Aurora DSQL application. Click **Next** to move on to the next section.

4. Programming with Aurora DSQL

Amazon Aurora DSQL is a PostgreSQL-compatible relational database. It uses

the PostgreSQL wire protocol and database drivers, making it easy to use for software developers accustomed to building applications for relational databases. However, Aurora DSQL does introduce some behavioral differences that software developers will need to be aware of as they build applications that use Aurora DSQL.
We'll explore some of those differences in this module with a simple database client written in Java. You won't have to write any software code. Instead, we've provided it for you to run in the CloudShell environment. You'll see how to fetch an IAM authentication token and establish a JDBC connection to the database. Then you'll learn how to identify and respond to concurrency conflicts, implement transaction retries, and implement exponential backoff and jitter to reduce the chance of recolliding with other transactions. The patterns you'll learn in this section can be implemented in any programming language.
Click **Next** to continue.

Writing an Aurora DSQL client

A skeleton Java project called "hello-aurora-dsql" has been provided to you in the CloudShell environment in us-east-1. Switch over to the browser tab or window that has the AWS Management Console for us-east-1 and open the CloudShell terminal.
Let's verify that our Java project is setup correctly by running it. In the CloudShell terminal, enter hello1.sh. This is one of several convenience scripts that invoke the Maven build system to download any necessary dependencies, compile our programs, and execute them. You should see "Hello, Aurora DSQL" in the program output.

```
$ hello1.sh
Hello, Aurora DSQL
$
```

Now let's look at a basic Java code template for connecting to Aurora DSQL and executing a query. We'll walk through the important parts below and then we'll run it using a convenience script.

```
package software.amazon.dsql;

import org.postgresql.jdbc.SslMode;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import
software.amazon.awssdk.regions.providers.DefaultAwsRegionProviderChain;
import software.amazon.awssdk.services.dsql.DsqlUtilities;

import java.sql.*;
import java.util.Properties;


public class HelloDSQL2 {
    public static void main(String[] args) {

        if (args.length == 0) {
            System.err.println("Cluster endpoint URL is required");
            System.exit(-1);
        }

        String dbEndpoint = args[0];

        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Unable to load PostgreSQL driver", e);
        }

        //
        // Our "password" is an IAM authentication token.  We generate a token
using the Aurora DSQL SDK.
        //
        DsqlUtilities utilities = DsqlUtilities.builder()
                .region(DefaultAwsRegionProviderChain.builder().build().getRegion())
                .credentialsProvider(DefaultCredentialsProvider.create()).build();

        String password = utilities.generateDbConnectAdminAuthToken(builder ->
builder.hostname(dbEndpoint));

        Properties props = new Properties();
        props.setProperty("user", "admin");
        props.setProperty("password", password);
        props.setProperty("sslmode", SslMode.REQUIRE.name());

        String jdbcUrl = String.format("jdbc:postgresql://%s:5432/postgres",
dbEndpoint);
        try (Connection conn = DriverManager.getConnection(jdbcUrl, props);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("select username, first_name,
last_name from xpoints.customers limit 5;")) {
            while (rs.next()) {
                System.out.println(rs.getString("username") + ": " +
rs.getString("first_name") + " " + rs.getString("last_name"));
            }
        } catch (SQLException e) {
            System.err.println("MESSAGE:    " + e.getMessage());
            System.err.println("ERROR CODE: " + e.getErrorCode());
            System.err.println("SQL STATE:  " + e.getSQLState());
        }
    }
}
```

The first important thing our code does is load the PostgreSQL JDBC driver class. This is just the standard open-source PostgreSQL driver and is not specific to Aurora DSQL or even Aurora in general.

```
try {
    Class.forName("org.postgresql.Driver");
} catch (ClassNotFoundException e) {
    throw new RuntimeException("Unable to load PostgreSQL driver", e);
}
```

Aurora DSQL does not support password authentication. Instead, we authenticate using an IAM token. We'll fetch the IAM token using the AWS SDK for Aurora DSQL and pass that into our PostgreSQL JDBC driver as the password. The code is listed below. Note that we're fetching a token for the database admin for convenience. To fetch a token for a custom database user,
call generateDbConnectAuthToken() on DsqlUtilities in stead of generateDbConnectAdminAuthToken() and be sure to use the correct username when establishing the database connection. You'd also need the proper IAM permissions to actually connect to the database as that user. We'll proceed with connecting as the

database admin.

```
DsqlUtilities utilities = DsqlUtilities.builder()
        .region(DefaultAwsRegionProviderChain.builder().build().getRegion())
        .credentialsProvider(DefaultCredentialsProvider.create()).build();

String password = utilities.generateDbConnectAdminAuthToken(builder ->
builder.hostname(dbEndpoint));
```

The rest of the code is boilerplate JDBC code. We setup our connection parameters in a Properties object and pass it into the DriverManager to establish the connection. We execute a simple SELECT query to fetch several records from our application's xpoints.customers table and print them to the console. We catch SQLException for any database-related problems and display the exception info to the console.

```
Properties props = new Properties();
props.setProperty("user", "admin");
props.setProperty("password", password);
props.setProperty("sslmode", SslMode.REQUIRE.name());

String jdbcUrl = String.format("jdbc:postgresql://%s:5432/postgres",
dbEndpoint);
```

```
try (Connection conn = DriverManager.getConnection(jdbcUrl, props);
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select username, first_name,
last_name from xpoints.customers limit 5;")) {
   while (rs.next()) {
      System.out.println(rs.getString("username") + ": " +
rs.getString("first_name") + " " + rs.getString("last_name"));
   }
} catch (SQLException e) {
   System.err.println("MESSAGE:    " + e.getMessage());
   System.err.println("ERROR CODE: " + e.getErrorCode());
   System.err.println("SQL STATE:  " + e.getSQLState());
}
```

To run the code, you'll need the endpoint URL of your Aurora DSQL cluster for us-east-1. You can find that information in the Aurora DSQL page of the AWS Management Console in us-east-1 as shown in Create a multi-region Aurora DSQL cluster. Once you have that, run the following statement, replacing <<YOUR CLUSTER ENDPOINT>> with your cluster's us-east-1 endpoint.

```
hello2.sh <<YOUR CLUSTER ENDPOINT>>
```

You should see a list of customers from your database. The exact customers in your output might be different than shown here.

```
$ hello2.sh <<YOUR CLUSTER ENDPOINT>>
carlos_salazar_121@example.mil: Carlos Salazar
kwaku_mensah_108@example.net: Kwaku Mensah
efua_owusu_152@example.org: Efua Owusu
nikki_wolf_132@example.edu: Nikki Wolf
nikhil_jayashankar_17@example.edu: Nikhil Jayashankar
```

This section covered connection basics. In the next section, we'll look at how to program transactional logic for Aurora DSQL optimistic concurrency, which might be a little different than what you're used to. Click Next to continue.

Handling concurrency conflicts

In Working with Aurora DSQL Concurrency, we learned about Aurora DSQL's optimistic concurrency behaviors by executing conflicting transactions in psql. We learned that with optimistic concurrency, the best approach is to retry your transaction in the

event of a concurrency error. In this section, we'll see how to put that into practice in software code. We'll modify our HelloDSQL program to perform an update transaction. Our program will detect concurrency collisions and retry the transaction. We'll implement exponential backoff and jitter to reduce the likelihood of subsequent collisions. We'll run the program and induce a concurrency collision to see it in action.

What does it mean to retry a transaction? During a retry, the entire transaction should be re-executed, including all query statements and the application logic for the transaction. This ensures that your application performs all of the same steps and decision logic, but with the current state of the data. Application logic executed in the scope of the database transaction should be *idempotent*. That is, it must be repeatable without causing side effects.

What is an example of a side-effect? Imagine we have a process that starts a transaction, does some database work, puts a message on an Amazon Simple Queue Service (SQS) queue to trigger the next step in the workflow, and then commits. If the transaction fails, we run the logic again, putting *another* message on the queue, triggering the next step in the workflow again. That can't be good! Or what if, on the second attempt, we make a different decision based on the latest state of the data and decide that the next step in the workflow shouldn't be triggered? It's too late! We already did it. This is an example of a side-effect.

Now let's look at some Java code that implements retries on concurrency conflict. Some of the more interesting bits are highlighted for your attention. We'll walk through the important parts below and then we'll run it using a convenience script.

```
package software.amazon.dsql;

import org.postgresql.jdbc.SslMode;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import
software.amazon.awssdk.regions.providers.DefaultAwsRegionProviderChain;
```

```
import software.amazon.awssdk.services.dsql.DsqlUtilities;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Properties;
import java.util.UUID;


public class HelloDSQL3 {
    private static final double JITTER_BASE = 20d;
    private static final double JITTER_MAX = 1000 * 5d;


    private static void backoff(int attempt) {
        long duration = (long) (Math.min(JITTER_MAX, JITTER_BASE *
Math.pow(2.0d, attempt)) * Math.random());
        try {Thread.sleep(duration);} catch (InterruptedException ignored) {}
    }


    public static void main(String[] args) {

        if (args.length == 0) {
            System.err.println("Cluster endpoint URL is required");
            System.exit(-1);
        }

        String dbEndpoint = args[0];

        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Unable to load PostgreSQL driver", e);
        }

        DsqlUtilities utilities = DsqlUtilities.builder()
                .region(DefaultAwsRegionProviderChain.builder().build().getRegion())
                .credentialsProvider(DefaultCredentialsProvider.create()).build();

        String password = utilities.generateDbConnectAdminAuthToken(builder ->
builder.hostname(dbEndpoint));

        Properties props = new Properties();
        props.setProperty("user", "admin");
        props.setProperty("password", password);
        props.setProperty("sslmode", SslMode.REQUIRE.name());

        String jdbcUrl = String.format("jdbc:postgresql://%s:5432/postgres",
dbEndpoint);
        try (Connection conn = DriverManager.getConnection(jdbcUrl, props);
             PreparedStatement stmt = conn.prepareStatement("update
xpoints.customers set age = ? where id = ?");
             AutoCloseable cleanup = conn::rollback) {

            conn.setAutoCommit(false);

            int attempt = 0;
            while (attempt++ < 5) {
                if (attempt > 1)
                    backoff(attempt);

                try {
                    System.out.println("Attempt #" + attempt);
                    stmt.setInt(1, 40);
                    stmt.setObject(2, UUID.fromString("02bd4416-0759-4763-
b256-2d97dccf37aa"));
                    stmt.executeUpdate();

                    //
                    // Do all sorts of interesting business logic here!
                    //

                    System.out.println("Sleeping for 15 seconds before committing...");
                    try {Thread.sleep(15000);} catch (InterruptedException ignored) {}

                    conn.commit();
                    System.out.println("Successful commit!");
                    attempt = 5 + 1; // Force attempts above the max
                } catch (SQLException e) {
                    try {conn.rollback();} catch (SQLException ignored) {}

                    if ("40001".equals(e.getSQLState())) {
                        System.err.println("Concurrency collision!");
                        System.err.println();
                    }

                    if (attempt == 5 || !"40001".equals(e.getSQLState())) {
                        throw e;
                    }
                }
            }
        } catch (SQLException e) {
            System.err.println("MESSAGE:    " + e.getMessage());
            System.err.println("ERROR CODE: " + e.getErrorCode());
            System.err.println("SQL STATE:  " + e.getSQLState());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Lines 29-53 repeat the boilerplate JDBC code that we saw in Programming with Aurora DSQL, so we won't discuss these. Lines 55-58 are different than before. We setup a PreparedStatement for our UPDATE query and turn off auto-commit so we can demarcate the transaction boundaries ourselves.

```
54
55
56
57
58
        try (Connection conn = DriverManager.getConnection(jdbcUrl, props);
             PreparedStatement stmt = conn.prepareStatement("update
xpoints.customers set age = ? where id = ?");
             AutoCloseable cleanup = conn::rollback;) {

            conn.setAutoCommit(false);
```

Next we setup our retry loop. The attempt variable tracks how many times we've tried our transaction. We execute our transaction logic in a while loop, setting a maximum of 5 attempts. We increment the attempt count in the while loop.

Next, if this is not our first attempt at this transaction, we call our backoff() method to wait a little bit. We'll talk about the implementation of backoff() in a bit. For now, understand that this helps to avoid cases where multiple database clients try to update the same data, then quickly retry the transaction several times as fast as possible, colliding with each other again each time.

Lines 71-73 are a bit of mystical hand-waving to indicate that you can do other business logic in your transaction loop. You're not limited to just database interaction logic. You should re-execute any business logic that makes decisions or provides results based on the data you're reading or writing in the database. This way, you'll always be making decisions based on the latest valid state of the data.

Finally, we attempt to commit our transaction. At this point,

Aurora DSQL will check to see if any other transaction in this region or the other has updated the data we're trying to update. If there are no conflicting transactions, our transaction succeeds and we increment the attempt count beyond the maximum to end our loop.

```
        int attempt = 0;
        while (attempt++ < 5) {
            if (attempt > 1)
                backoff(attempt);

            try {
                System.out.println("Attempt #" + attempt);
                stmt.setInt(1, 40);
                stmt.setObject(2, UUID.fromString("02bd4416-0759-4763-b256-2d97dccf37aa"));
                stmt.executeUpdate();

                //
                // Do all sorts of interesting business logic here!
                //

                System.out.println("Sleeping for 15 seconds before committing...");
                try {Thread.sleep(15000);} catch (InterruptedException ignored) {}

                conn.commit();
                System.out.println("Successful commit!");
                attempt = 5 + 1; // Force attempts above the max
```

However, if there WAS a conflicting transaction that committed before ours, we'll get an SQLException.

The first thing we do is roll the transaction back. Next, we need to identify if this exception was caused by a concurrency conflict. Aurora DSQL throws an error with a SQL state of 40001, which is a standard PostgreSQL error code

indicating a serialization failure. If our exception is a concurrency conflict and we have not exhausted our retries, we simply continue into the next iteration of our while loop. If this is not a concurrency exception or we've exhausted our maximum retries, then we'll re-throw the error. For the sake of this workshop, we catch and log the error in the last part of the program (see lines 93-99 above).

```
            } catch (SQLException e) {
                try {conn.rollback();} catch (SQLException ignored) {}

                if ("40001".equals(e.getSQLState())) {
                    System.err.println("Concurrency collision!");
                    System.err.println();
                }

                if (attempt == 5 || !"40001".equals(e.getSQLState())) {
                    throw e;
                }
            }
```

Now that we've seen how to catch concurrency conflicts and implement retries, let's dig a little deeper into our backoff() method.

In the event of concurrency conflicts, we don't want to simply retry our transaction in a tight loop. If other database clients do the same thing, we'll continue to collide, decreasing the chance

that we'll be able to commit before the maximum number of retries. To avoid this, we'll add exponential back-off on retries to add a small but progressively increasing wait before retrying. This will reduce contention on the row we're trying to update. We'll also add a bit of "jitter", or randomness, to the wait time to further reduce the chance of re-colliding with other clients.

We start with a base amount of wait time, stored in the JITTER_BASE variable. We multiple this number by 2 to a power that increases with each attempt, exponentially increasing our wait time with each attempt. This is our exponential backoff. Now we add a bit of randomness by multiplying that calculated wait time by a randomly generated number between 0.0 and 1.0. Finally, we make sure our wait time doesn't go above a maximum threshold indicated by the JITTER_MAX variable. Now that we've calculated our wait time, we wait! When we've slept for our "backoff" period, we return so that our transaction logic can resume.

```
   private static void backoff(int attempt) {
       long duration = (long) (Math.min(JITTER_MAX, JITTER_BASE *
Math.pow(2.0d, attempt)) * Math.random());
       try {Thread.sleep(duration);} catch (InterruptedException ignored) {}
   }
```

There are several approaches to implementing exponential backoff and jitter. To learn more, see Exponential Backoff and Jitter

and check out this blog post

on Aurora DSQL concurrency.

Now let's run our code.

Run the code

You should have a browser tab or window opened to the AWS Console in the us-east-1 region and one in the us-east-2 region. Each tab or window should have a CloudShell session open. We'll use the browser tab in us-east-1 to run our program and the browser tab in us-east-2 to query Aurora DSQL.

To run the code, you'll need the endpoint URL of your Aurora DSQL cluster for us-east-1. You can find that information in the Aurora DSQL page of the AWS Management Console in us-east-1 as shown in Create a multi-region Aurora DSQL cluster.

Once you have that, run the following statement, replacing <<YOUR CLUSTER ENDPOINT>> with your cluster's us-east-1 endpoint.

```
hello3.sh <<YOUR CLUSTER ENDPOINT>>
```

You should see the following output:

```
$ ./hello3.sh <<YOUR CLUSTER ENDPOINT>>
Attempt #1
Sleeping for 15 seconds before committing...
Successful commit!
```

Our program completed successfully, which isn't very exciting. Now let's induce a concurency conflict. Our program updates the xpoints.customers record with an ID of 02bd4416-0759-4763-b256-2d97dccf37aa, belonging to Jonn Doe. We'll modify that record while our program is running.

In your browser tab for us-east-2, connect to Aurora DSQL as the admin user. Instructions for connecting to Aurora DSQL can be found in Connect to Aurora DSQL. You can also use the psqli.sh convenience script. In the us-east-2 tab, run the command below, replacing "<<YOUR CLUSTER ENDPOINT>>" with the us-east-2 endpoint for your Aurora DSQL cluster.

```
psqli.sh <<YOUR CLUSTER ENDPOINT>>
```

Execute the following query to find John Doe's age:

```
select first_name, last_name, age from xpoints.customers where id =
'02bd4416-0759-4763-b256-2d97dccf37aa';

 first_name | last_name | age
------------+-----------+-----
 John       | Doe       |  40
(1 row)
```

You should now be ready to run our program in us-east-1 and quickly execute a query in psql in us-east-2.

Execute the code again in us-east-1 by running the following script, replacing "<<YOUR CLUSTER ENDPOINT>>" with your cluster's us-east-1 endpoint.

```
hello3.sh <<YOUR CLUSTER ENDPOINT>>

$ ./hello3.sh <<YOUR CLUSTER ENDPOINT>>
Attempt #1
Sleeping for 15 seconds before committing...
```

When you see the "Sleeping" message, quickly switch over to the psql session in us-east-2 and run the following query:

```
update xpoints.customers set age = age + 1 where id = '02bd4416-0759-4763-
```

```
b256-2d97dccf37aa';
```

Your output should look like this:

```
$ ./hello3.sh <<YOUR CLUSTER ENDPOINT>>
Attempt #1
Sleeping for 15 seconds before committing...
Concurrency collision!

Attempt #2
Sleeping for 15 seconds before committing...
Successful commit!
```

You induced a concurrency collision at the first transaction attempt. The transaction was able to complete on the second attempt. Run the code again as shown below, replacing "<<YOUR CLUSTER ENDPOINT>>" with your cluster's us-east-1 endpoint.

```
hello3.sh <<YOUR CLUSTER ENDPOINT>>
```

When you see the "Sleeping" message, quickly switch over to the psql session in us-east-2 and run the following query:

```
update xpoints.customers set age = age + 1 where id = '02bd4416-0759-4763-
b256-2d97dccf37aa';
```

Keep an eye on our running program and re-run the UPDATE query in us-east-2 whenever you see the "Sleeping" message in us-east-1. After the 5th attempt, the program should fail. The output should look like this:

```
$ ./hello3.sh <<YOUR CLUSTER ENDPOINT>>
Attempt #1
Sleeping for 15 seconds before committing...
Concurrency collision!

Attempt #2
Sleeping for 15 seconds before committing...
Concurrency collision!

Attempt #3
Sleeping for 15 seconds before committing...
Concurrency collision!

Attempt #4
Sleeping for 15 seconds before committing...
Concurrency collision!

Attempt #5
Sleeping for 15 seconds before committing...
Concurrency collision!
```

```
MESSAGE:    ERROR: change conflicts with another transaction, please retry:
(OC000)
ERROR CODE: 0
SQL STATE:  40001
```

If you query John Doe's age, it won't be the value (40) that the program is trying to set:

```
postgres=> select age from xpoints.customers where id =
'02bd4416-0759-4763-b256-2d97dccf37aa';
 age
-----
  45
(1 row)
```

Feel free to experiment with this exercise, changing the number of concurrency collisions to observe the program's behavior.

To re-cap, we attempted a write in us-east-1 using our program and we made a conflicting write in us-east-2. The first write to commit won and the second write *in a different region* **was notified of the conflict. That's pretty cool!**

Our program demonstrates how to detect Aurora DSQL concurrency conflicts in code, and how to implement transaction retries with exponential backoff and jitter to reduce subsequent collisions. Adapt this logic to your code when building applications with Aurora DSQL!

Congratulations! You've learned how to detect concurrency conflicts in software code and how to implement transaction retries with exponential backoff and jitter. Click Next to continue to the next section.

(OPTIONAL) Explore the Rewards App Code

Now that you've seen an example of implementing transaction retry logic for Aurora DSQL, feel free to explore the Rewards Points application code to see more realistic examples.

The /home/cloudshell-user/rewards-backend directory in the CloudShell environment in **us-east-1** contains the Java project for our Rewards Points application. The src/main/java/software/amazon/dsql/rewards directory and its sub-directories within the project contain Java source code files for the AWS Lambda functions that serve our API. Feel free to look through them to see how the Rewards Points application was built, how it works with Aurora DSQL, how it executes transactions, and how it handles failures and retries.

To get to that directory, enter the following command in CloudShell:

```
cd /home/cloudshell-user/rewards-backend/src/main/java/software/amazon/
dsql/rewards
```

You can view the source code files using vi, nano, or simply use

the cat or more programs to scroll them to the terminal.
When you're done, click **Next** to continue to the next section.

5. Wrap-Up

Congratulations! You've reached the end of the workshop. You've seen how easy it is to create a multi-region cluster with Amazon Aurora DSQL's minimal cluster creation workflow. You've learned about Aurora DSQL's optimistic concurrency control model and you've seen how Aurora DSQL's bi-directional synchronous cross-region replication keeps regions strongly consistent.

6. Clean-Up

This workshop is delivered through an AWS hosted event where AWS accounts are provided to you. At the conclusion of the event, your entire account will be decommissioned. You do not need to clean-up AWS resources yourself and you will not incur any AWS utilization charges.