# PostgreSQL On-premises Migration to Azure

## Technical white paper

**Published:** June 2021
**Applies to:** Azure Database for PostgreSQL

Copyright 2021

# Table of Contents


Introduction ..... 8
- PostgreSQL ..... 8
- Azure Database for PostgreSQL ..... 8

Use Case ..... 10
- Overview ..... 10

Assessment ..... 12
- PostgreSQL Versions ..... 12
- Database data and objects ..... 13
- Limitations ..... 14
- Source Systems ..... 17
- Performance Analysis Tools ..... 18
- Azure Database for PostgreSQL - Service Tiers ..... 19
- WWI Use Case ..... 26
- Assessment Checklist ..... 26

Planning ..... 27
- Landing Zone ..... 27
- Networking ..... 27
- Private Link and/or VNet integration ..... 29
- Azure Arc ..... 29
- SSL/TLS Connectivity ..... 30
- WWI Use Case ..... 30
- Planning Checklist ..... 30

Migration Methods ..... 31
- PostgreSQL pgAdmin ..... 31
- Dump and restore (pg_dump/pg_dumpall) ..... 32
- Azure Data Factory (ADF) ..... 32
- Other open-source tools ..... 32
- Replication ..... 33
- Azure Database Migration Service (DMS) ..... 35
- Fastest/Minimum Downtime Migration ..... 36
- Decision Table ..... 37

WWI Use Case ..... 38
Migration Methods Checklist ..... 38
Test Plans ..... 39
Overview ..... 39
Sample Queries ..... 39
Rollback Strategies ..... 44
WWI Use Case ..... 45
Checklist ..... 45
Performance Baselines ..... 46
Tools ..... 46
Server Parameters ..... 46
WWI Use Case ..... 48
Data Migration with Backup and Restore ..... 50
Setup ..... 50
Configure Server Parameters ..... 50
Data ..... 50
Revert Server Parameters ..... 55
Summary - Backup and Restore ..... 55
Data Migration with PostgreSQL COPY Command ..... 56
Setup ..... 56
Configure Server Parameters ..... 56
Install PgBouncer ..... 56
Schema ..... 56
Data ..... 56
Revert Server Parameters ..... 59
Data Migration with Azure Data Factory (ADF) ..... 60
Setup ..... 60
Configure Server Parameters ..... 60
Install PgBouncer ..... 60
Configure Network Connectivity ..... 60
Schema ..... 60
Data ..... 60

Revert Server Parameters ..... 76
Data Migration with Azure Database Migration Service (DMS) ..... 77
Setup ..... 77
Configure Server Parameters ..... 77
Install PgBouncer ..... 77
Configure Network Connectivity ..... 77
Schema ..... 77
Data ..... 77
Revert Server Parameters ..... 82
Setup ..... 83
Replication setup ..... 83
Data Migration with PostgreSQL Logical Decoding ..... 86
Setup ..... 86
wal2json ..... 88
Replication setup ..... 88
Setup ..... 93
Configure Server Parameters ..... 93
Prerequisites ..... 93
Schema ..... 93
Data ..... 93
Revert Server Parameters ..... 99
Data Migration ..... 100
Back up the database ..... 100
Offline vs. Online ..... 100
Data Drift ..... 100
Performance recommendations ..... 101
Performing the Migration Checklist ..... 103
Common Steps ..... 104
Migrate to the latest PostgreSQL version ..... 104
WWI Use Case ..... 104
Database Objects ..... 105
Execute migration ..... 109

Data Migration Checklist ..... 110
Data Migration - Schema ..... 111
Schema Migration ..... 111
Data Migration - Server Parameters - Ingress ..... 121
Configuring Server Parameters (Source) ..... 121
Configuring Server Parameters (Target) ..... 121
Exporting Server Parameters ..... 121
Configure Egress Server Parameters (Source) ..... 123
Configure Ingress Server Parameters (Target) ..... 124
Data Migration - Revert Server Parameters ..... 126
Configure Server Parameters ..... 126
Data Migration - Application Settings ..... 127
Setup ..... 127
Update Applications to support SSL ..... 127
Change Connection String for the Java API ..... 127
Post Migration Management ..... 128
Monitoring and Alerts ..... 128
Server Parameters ..... 129
PowerShell Module ..... 130
Azure Database for PostgreSQL Upgrade Process ..... 130
WWI Use Case ..... 130
Management Checklist ..... 131
Optimization ..... 131
Monitoring Hardware and Query Performance ..... 131
Typical Workloads and Performance Issues ..... 132
Query Performance Insight (QPI) ..... 132
Query Store (QS) ..... 132
Performance Recommendations ..... 135
Azure Advisor ..... 135
Upgrading the Tier ..... 135
Scale the Server ..... 135
Moving Regions ..... 135

Quick Tips ... 136
WWI Use Case ... 136
Optimization Checklist ... 136
Business Continuity and Disaster Recovery (BCDR) ... 138
Backup and Restore ... 138
Replicas ... 139
Deleted Servers ... 139
Regional Failure ... 139
WWI Use Case ... 140
BCDR Checklist ... 141
Security ... 143
Authentication ... 143
Threat Protection ... 143
Audit Logging ... 143
Encryption ... 144
Firewall ... 144
Private Link ... 144
Security baseline ... 145
Security Checklist ... 145
Summary ... 146
Questions and Feedback ... 146
Find a partner to assist in migrating ... 146
Appendix A: Environment Setup ... 148
Deploy the ARM template ... 148
Open the Azure VM Ports ... 150
Allow Azure PostgreSQL Access ... 150
Connect to the Azure VM ... 150
Install Chrome ... 151
Install ActivePerl 5.26 (PostgreSQL 10.16) ... 151
Install Python 3.7.4 (PostgreSQL 10.16) ... 151
Setup Java and Maven ... 152
Install PostgreSQL 10.16 ... 154

Configure Language Packs ..... 156
pgAdmin ..... 156
Azure Data Studio ..... 156
Download artifacts ..... 157
Deploy the Database ..... 157
Configure Blob Data ..... 160
Install Azure CLI ..... 161
Install NodeJS ..... 161
Install and Configure Visual Studio Code ..... 161
Configure the Web Application API ..... 161
Test the Web Application ..... 164
Configure the Web Application Client ..... 164
Deploy the Java Server Application to Azure ..... 164
Deploy the Angular Web Application to Azure ..... 168
Configure Network Security (Secure path) ..... 171
Appendix B: ARM Templates ..... 172
Secure ..... 172
Non-Secure ..... 172
Appendix B: Citus (Multi-Tenant) App Configuration ..... 173
Setup ..... 173
App Comparison ..... 175
Run the Sample App Locally ..... 177
Data Migration ..... 179
More Citus Modifications ..... 179
Migrate App to Azure and Test ..... 180
Appendix C: Default server parameters PostgreSQL 10.0 ..... 184
Appendix D: Default server parameters Azure (Single Server) V11 ..... 191
Configure PostgreSQL SSL ..... 199
Install OpenSSL ..... 199
Create a server certificate ..... 199
Configure the pg_hba.conf file ..... 200
Restart the server ..... 200

Configure PgBouncer ..... 202
Create SSL Certs ..... 202
Install PgBouncer ..... 202
Configure pgbouncer.ini file ..... 203
Test PgBouncer ..... 204

# Introduction

This migration guide is designed to provide snackable and actionable information for PostgreSQL customers and software integrators seeking to migrate PostgreSQL workloads to Azure Database for PostgreSQL. This guide will provide realistic guidance planning for executing a majority of the PostgreSQL migrations to Azure.

Workload functionality and existing application connectivity can present challenges when migrating existing PostgreSQL databases to the cloud.

The guide offers helpful links and recommendations focusing on migrating databases, ensuring performance, and functional application operational parity.

The information provided will center on a customer journey using the Microsoft Cloud Adoption Framework to perform assessment, migration, and post-optimization activities for an Azure Database for PostgreSQL environment.

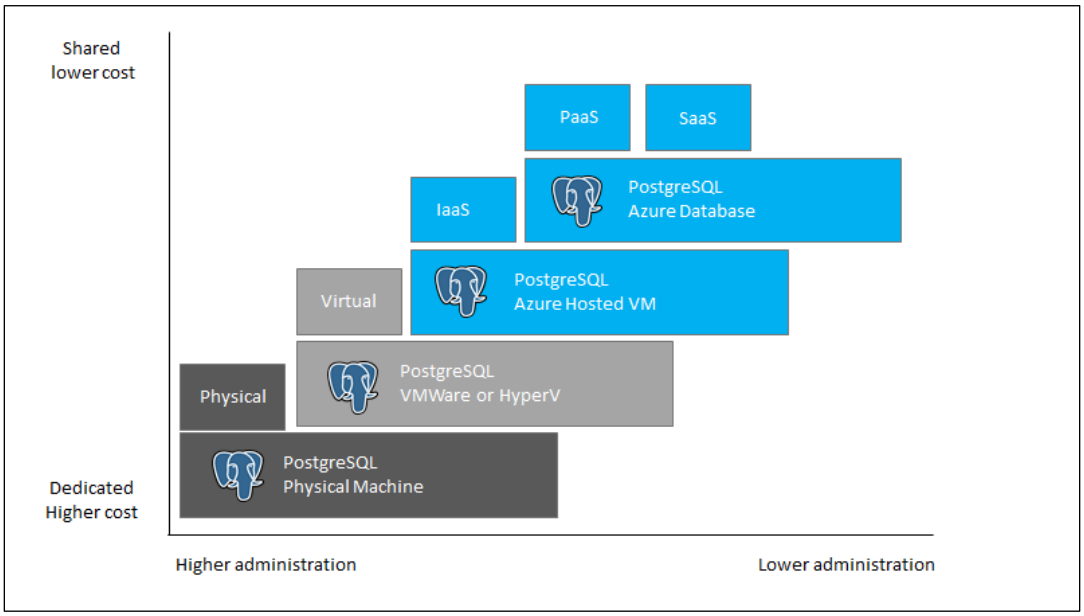
## PostgreSQL

PostgreSQL has a rich history in the open source community and is heavily used in corporate websites and critical applications. This guide will assist administrators asked to scope, plan, and execute a migration. New PostgreSQL Administrators should review the PostgreSQL Documentation for a deeper information of the internal workings on PostgreSQL. Additionally, each document section contains links to helpful reference articles and tutorials.

## Azure Database for PostgreSQL

Microsoft offers a fully managed PostgreSQL environment as a Platform as a Service (PaaS) Azure Database for PostgreSQL. In this fully managed environment, the operating system and software updates are automatically applied, as well as the implementation of high availability and protection of the data.

PostgreSQL can also run in Azure VMs. Reference the Choose the right PostgreSQL Server option in Azure article for the deployment type options appropriate for the target data workload.




Comparison of PostgreSQL environments.

This guide will focus entirely on migrating the on-premises PostgreSQL workloads to the Platform as a Service (PaaS) Azure Database for PostgreSQL (Single server, Flexible server, and Hyperscale Citus) offering due to its various advantages over Infrastructure as a Service (IaaS) such as scale-up and scale-out, pay-as-you-go, high availability, security and manageability features.

# Use Case

## Overview

World Wide Importers (WWI) is a San Francisco, California-based manufacturer and wholesale distributor of novelty goods. They began operations in 2002 and developed an effective business-to-business (B2B) model, selling the items they produce directly to retail customers throughout the United States. Its customers include specialty stores, supermarkets, computing stores, tourist attraction shops, and some individuals. This B2B model enables a streamlined distribution system of their products, allowing them to reduce costs and offer more competitive pricing on their manufactured items. They also sell to other wholesalers via a network of agents who promote their products on WWI's behalf.

Before launching into new areas, WWI wants to ensure its IT infrastructure can handle the expected growth. WWI currently hosts all its IT infrastructure on-premises at its corporate headquarters and believes moving these resources to the cloud enables future growth. They have tasked their CIO with overseeing the migration of their customer portal and the associated data workloads to the cloud.

WWI would like to continue to take advantage of the many advanced capabilities available in the cloud, and they are interested in migrating their databases and associated workloads into Azure. They want to do this quickly and without having to make any changes to their applications or databases. Initially, they plan on migrating their Java-based customer portal web application and the associated PostgreSQL databases and workloads to the cloud.

### Migration Goals

The primary goals for migrating their PostgreSQL databases and associated workloads to the cloud include:

- Improve their overall security posture by encrypting data at rest and in-transit.
- Enhance the high availability and disaster recovery (HA/DR) capabilities.
- Position the organization to leverage cloud-native capabilities and technologies such as point-in-time restore.
- Take advantage of administrative and performance optimization features of Azure Database for PostgreSQL.
- Create a scalable platform that they can use to expand their business into more geographic regions.
- Allow for enhanced compliance with various legal requirements where PII information is stored.

WWI used the Cloud Adoption Framework (CAF) to educate their team on following best practices guidelines for cloud migration. Using CAF as a higher-level migration guide, WWI customized their migration into three main stages. They defined activities to be addressed within each stage to ensure a successful lift and shift cloud migration.

These stages include:

| Stage | Name | Activities |
|---|---|---|
| **1** | Pre-migration | Assessment, Planning, Migration Method Evaluation, Application Implications, Test Plans, Performance Baselines |
| **2** | Migration | Execute Migration, Execute Test Plans |
| **3** | Post-migration | Business Continuity, Disaster Recovery, Management, Security, Performance Optimization, Platform modernization |

WWI has several instances of PostgreSQL running with varying versions ranging from 9.5 to 11. They would like to move their older instances to the latest PostgreSQL version as soon as possible, but there are some concerns regarding applications issues. A decision has been made to migrate to the cloud first and upgrade the PostgreSQL version later, knowing that PostgreSQL 9.5 is deprecated and 9.6 will be deprecated in November 2021.

They would also like to ensure that their data workloads are safe and available across multiple geographic regions in case of failure and are looking at the available configuration options.

WWI wants to start off with a simple application for the first migration, and then move to more business-critical applications in a later phase. This strategy will provide the team with the knowledge and experience they need to prepare and plan for those future migrations.

# Assessment

Before jumping right into migrating a PostgreSQL workload, a fair amount of due diligence must be performed. This includes analyzing the data, hosting environment, and application workloads to validate the Azure landing zone is properly configured and prepared to host the soon-to-be migrated workloads.

## PostgreSQL Versions

PostgreSQL has a rich history starting in 1996 and was previously known as Ingres. Since then, it has evolved into a widely used database management system. Azure Database for PostgreSQL started with the support of PostgreSQL version 9.5 (retired as of 2/2021) and has continued to 13.0 (as of 4/2021). For a listing of all PostgreSQL versions, reference this detailed page.

For the latest on Azure Database for PostgreSQL version support, reference Supported Azure Database for PostgreSQL server versions. In the Post Migration Management section, we will review how upgrades (such as 9.6.1 to 9.6.16) are applied to the PostgreSQL instances in Azure.

**Note**: PostgreSQL 9.5 is already end-of-life (EOL). PostgreSQL 9.6 will be EOL in November 2021.

Knowing the source PostgreSQL version is important. The applications using the system may be using database objects and features that are specific to that version. Reference the PostgreSQL Feature Matrix for these major version feature differences. Migrating a database to a lower version could cause compatibility issues and loss of functionality. The data and application instances should be fully tested before migrating. Newer version features introduced or removed could break the application.

Although PostgreSQL has been great at keeping breaking changes at a minimum and keeping compatibility between versions, there are a handful of examples that may influence the migration path and version, or force application changes:

- Fully enforce uniqueness of table and domain constraint names
- Fix to_date(), to_number(), and to_timestamp() to skip a character for each template character
- Remove createlang and droplang command-line applications
- Remove support for version-0 function calling conventions
- Removal of support for older PostgreSQL syntax
- Older version going out of support

To check the PostgreSQL server version, run the following SQL command against the PostgreSQL instance:

```
SELECT version();
```

For a list of changes between versions, reference the latest release documentation:

- PostgreSQL 13
- PostgreSQL 12
- PostgreSQL 11
- PostgreSQL 10

**Hyperscale Citus**

In addition to the basic PostgreSQL editions above, there is also a Citus-based version of PostgreSQL. The Hyperscale (Citus) deployment option scales queries across multiple machines using sharding, to serve applications that require greater scale and performance. For more information, reference What is Azure Database for PostgreSQL - Hyperscale (Citus)? and some basic architectural concepts here.

## Database data and objects

Data is only one component of database migration. The database supporting objects must be migrated and validated to ensure the applications continue to run reliably. As part of the assessment, it is important to understand system features being used, other than data storage.

Here is a list of inventory items to be queried before and after the migration:

- Casts
- Collations
- Configuration settings
- Domains
- Extensions
- Foreign Data Wrappers
- Foreign Tables
- Full-Text Search (Configurations, Dictionaries, Parsers)
- Functions
- Indexes
- Languages
- Materialized View and Views

- Primary keys, foreign keys
- Procedures
- Rules
- Tables (schema)
- Trigger and Event Trigger functions

After reviewing the above items, notice there is much more than just data that may be required to migrate a PostgreSQL workload. The following sections below address more specific details about several of the above.

## Limitations

Azure Database for PostgreSQL is a fully supported version of PostgreSQL running as a platform as a service offering. However, there are some common limitations when performing an initial assessment.

Azure Database for PostgreSQL Single Server is a platform that manages patching, backups, and high-availability for PostgreSQL workloads. Azure Database for PostgreSQL Flexible Server provides these same benefits, while also supporting greater user configuration, such as additional high-availability offerings, pgBouncer connection pooling integration, and more. Lastly, the Hyperscale (Citus) offering provides horizontal scaling for queries. Microsoft has determined that customers benefit from using Hyperscale (Citus) for datasets larger than 100 GB.

**Note**: The Flexible Server offering is in preview.

Microsoft provides these offerings through an architecture that decouples compute and storage, allowing for features such as automatic failover, without requiring application changes.

In addition to the common limitations, each service has its limitations:

- Single Server limitations
  - No automated upgrades between major database engine versions.
  - Storage size cannot be reduced. Modifications require creating a new server with desired storage size.
  - In some scenarios, UTF-8 characters are not fully supported in open-source PostgreSQL on Windows.
- Flexible Server limitations
  - Upgrades between major database engine versions are not automated.
  - Server storage can only be scaled in 2x increments.

  - Storage size cannot be reduced. Modifications require creating a new server with desired storage size.
  - PostgreSQL 10 and older are not supported.
  - Extension support is currently limited to certain Postgres extensions. Reference Extensions for more details.
- Hyperscale Citus Server limitations
  - Storage on the coordinator and worker nodes can be scaled up (increased) but can't be scaled down (decreased).
  - Up to 2 TiB of storage is supported on the coordinator and worker nodes.

Many of the other items are simply operational aspects that administrators should become familiar with as part of the operational data workload lifecycle management. This guide will explore many of these operational aspects in the Post Migration Management section.

### Full-Text Search

PostgreSQL has included a full-text search capability in various forms over the years. The most recent advances came in 9.6 with the GIN and GiST index types. Verify the instance has expanded the default full-text search to include custom configurations, dictionaries, and parsers, and whether any applications are using them. Where supported, it will be necessary to ensure they are migrated to the target instance.

Searches rely on indexes. It will take time to rebuild them after the migration. Script this task out via SQL statements. Factor these task efforts into the migration plan.

Normal users cannot create full-text indexes. Full-text Search creation requires the superuser permission. Running as this user is not allowed in Azure Database for PostgreSQL, so plan accordingly.

### Languages

PostgreSQL supports the ability to define other languages for custom functions and stored procedures. Azure Database for PostgreSQL only supports plpgsql. If the database has any functions or procedures that utilize other languages, the only target migration option will be the IaaS (Virtual Machine) option.

### User-Defined Functions and Types (C/C++)

PostgreSQL allows functions to call external code such a C/C++ libraries as well as the definition of custom types using C/C++.

These functions (UDFs) can be identified in the instance by running the following query:

```
select n.nspname as function_schema,
      p.proname as function_name,
      l.lanname as function_language,
      case when l.lanname = 'internal' then p.prosrc
          else pg_get_functiondef(p.oid)
          end as definition,
      pg_get_function_arguments(p.oid) as function_arguments,
      t.typname as return_type
from pg_proc p
left join pg_namespace n on p.pronamespace = n.oid
left join pg_language l on p.prolang = l.oid
left join pg_type t on t.oid = p.prorettype
where n.nspname not in ('pg_catalog', 'information_schema')
order by function_schema,
        function_name;
```

Look for the function_language to be C or anything that is not plpgsql in the search results.

Production environments that require these types of functions will need to migrate to PostgreSQL on Azure Virtual Machines. Custom DLLs/object files are not supported on Azure instances.

### Extensions

Azure Database for PostgreSQL does not support the full range of **on-premises PostgreSQL** extensions. For the latest list, run the SELECT * FROM pg_available_extensions; script or see the latest documentation for each service:

- Single Server - (https://docs.microsoft.com/en-us/azure/postgresql/concepts-extensions).
- Flexible Server - (https://docs.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-extensions)
- Hyperscale Citus Server - (https://docs.microsoft.com/en-us/azure/postgresql/concepts-hyperscale-extensions)

Inventory your on-premises PostgreSQL extension requirements. Can the affected applications be upgraded to the Azure Database for PostgreSQL extensions? If not, the migration is limited to a PostgreSQL VM migration path.

### File System Writes

Any functions, stored procedures, or application code that execute queries (such as COPY) that need file system access are not allowed in Azure Database for PostgreSQL.

Review the application code to see if it makes any calls to the COPY command. Functions and stored procedures that contain the COPY command embedded can be exported.

### Hyperscale Citus Limitations

#### SQL

Since Citus provides distributed functionality by extending PostgreSQL, it uses the standard PostgreSQL SQL constructs. The vast majority of queries are supported, even when they combine data across the network from multiple database nodes. This support includes transactional semantics across nodes. All cross-node SQL is supported except the following:

- Correlated subqueries
- Recursive CTEs
- Table sample
- SELECT … FOR UPDATE
- Grouping sets

In order to perform a migration, identify any queries that use the above constructs and make modifications. There are various workarounds to referenced SQL queries in Workarounds in the Citus documentation.

#### Command Propagation

Running specific commands run on the coordinator node do not get propagated to the workers:

- CREATE ROLE/USER (gets propagated in Citus Enterprise)
- CREATE DATABASE
- ALTER … SET SCHEMA
- ALTER TABLE ALL IN TABLESPACE
- CREATE FUNCTION (use create_distributed_function)
- CREATE TABLE (see Table Types)

**Note**: View other FAQs about Hyperscale Citus here.

## Source Systems

The amount of migration preparation can vary depending on the source system and its location. Consider how to get the data from the source system to the target system. Migrating data can become challenging when firewalls and other networking components are between the source and target.

Internet migration speed is an important factor. Moving data over the Internet can be slower than using dedicated circuits to Azure. Consider setting up an ExpressRoute connection between the source network and the Azure network when moving many gigabytes, terabytes, and petabytes of data.

Do not overwhelm existing network infrastructure. If ExpressRoute is already present, the connection is likely being used by other applications. Performing a migration over an existing route can cause strain on the network throughput and potentially cause considerable performance degradation for both the migration and other applications using the network.

Lastly, disk space must be evaluated. When exporting a very large database, consider the size of the data. Ensure the system where the tool is running and the export location have enough disk space to perform the export operation. Performing a failed migration due to the lack of available disk space is a waste of time.

### Cloud Providers

Migrating databases from cloud services providers, such as Amazon Web Services (AWS), may require extra networking configuration steps to access the cloud-hosted PostgreSQL instances. Migration tools, like Azure Database Migration Service (DMS), require access from outside IP ranges and may be blocked. Work with your administrators to test connectivity before attempting a migration.

### On-premises

Like cloud provider-hosted environments, if the PostgreSQL data environment is behind corporate firewalls or other network security layers, a path will need to be created between the on-premises instance and Azure Database for PostgreSQL.

## Performance Analysis Tools

Many tools and methods can be used to assess the PostgreSQL data workloads and environments. Each tool will provide a different set of assessment and migration features and functionality. As part of this guide, we will review the most commonly used tools for assessing PostgreSQL data workloads.

### Azure Migrate

Although Azure Migrate does not support migrating PostgreSQL database workloads directly, it can be used when administrators are unsure of what users and applications are consuming the data, whether hosted in a virtual or hardware-based machine. Dependency analysis can be accomplished by installing and running the monitoring agent on the machine hosting the PostgreSQL workload. The agent will gather the useful information over a set period, such as a month. The dependency data can be analyzed to find **unknown connections** being made to the database. Connected application owners need to be notified of the pending migration.

In addition to the application dependency and user connectivity data, Azure Migrate can also be used to analyze the Hyper-V, VMWare, or physical servers to provide utilization patterns of the database workloads to help suggest the proper target environment.

### Telgraf for Linux

PostgreSQL installed on Linux can utilize the Microsoft Monitoring Agent (MMA) to gather data on virtual and physical machines. Additionally, consider using the Telegraf agent and its wide array of plugins to gather performance metrics.

For example, the Telegraf PostgreSQL integration uses built-in views to write to various destinations, including Azure Monitor.

## Azure Database for PostgreSQL - Service Tiers

Equipped with the assessment information (CPU, memory, storage, etc.), the migration user's next choice is to decide which Azure Database for PostgreSQL service and pricing tier option is required.

There are currently four potential options:

- Azure Database for PostgreSQL (VM)
- Azure Database for PostgreSQL (Single Server)
- Azure Database for PostgreSQL (Flexible Server)
- Azure Database for PostgreSQL (Hyperscale/Citus)

These options were discussed in the Limitations section of this document.

### Single Server Deployment Options

There are currently three pricing deployment options for the **Single Server** option:

- **Basic**: Workloads requiring light compute and I/O performance.
- **General Purpose**: Most business workloads requiring balanced compute and memory with scalable I/O throughput.
- **Memory-Optimized**: High-performance database workloads requiring in-memory performance for faster transaction processing and higher concurrency.

The deployment option decision can be influenced by the RTO and RPO requirements of the data workload. Data workloads over 4TB of storage require specific regions. Select a region that supports up to 16TB of storage.

**Note**: Contact the PostgreSQL team (AskAzureDBforPostgreSQL@service.microsoft.com) for regions that do not support the workload storage requirements.

Focus on the storage and Input/output Operations Per Second (IOPS) needs. The target system will always need at least as much storage as in the source system. Since IOPS are allocated 3/GB, it is important to match up the IOPS needs to the final storage size.

| Option | Factors |
|---|---|
| **Basic** | Development machine, no need for high performance with less than 1TB storage |
| **General Purpose** | Requirement for more IOPS than what the basic option can provide, but for storage less than 16TB |
| **Memory-Optimized** | Data workloads that utilize high memory or high cache and buffer related server configuration such as high concurrency |

### Flexible Server Deployment Options

There are currently three pricing deployment options for the **Flexible Server** option:

- **Burstable**: Best for workloads that don't need the full CPU continuously.
- **General Purpose**: Most business workloads requiring balanced compute and memory with scalable I/O throughput. Examples include servers for hosting web and mobile apps and other enterprise applications.
- **Memory-Optimized**: High-performance database workloads requiring in-memory performance for faster transaction processing and higher concurrency. Examples include servers for real-time data processing and high-performance transactional or analytical apps.

| Option | Factors |
|---|---|
| **Burstable** | Workloads that run periodically and for short periods with lower vCore requirements (up to 2) |
| **General Purpose** | Higher CPU needs (up to 64 vCores) that are consistent with up to 4GB of memory per vCore |
| **Memory-Optimized** | Data workloads that utilize high memory per vCore (up to 8GB per vCore) |

### Hyperscale Citus Deployment Options

There are currently two pricing deployment options for the **Hyperscale Citus Server** option:

- **Basic**: A single database node. Appropriate for smaller workloads in production.
- **Standard**: Contains coordinator node and at least two worker nodes. Scalable up to 20 worker nodes with 64 vCores, 8GB memory per vCore, and 2TB of storage per node.

Note that it is also possible to run Hyperscale (Citus) workloads using Azure Arc. Azure Arc extends the capabilities of Azure Resource Manager to on-premises and Kubernetes resources (even those in other clouds). Consult the documentation for Azure Arc-enabled Hyperscale (Citus) to understand the responsibilities of the customer and Microsoft in this model.

### Comparison of Services

Which Azure Database for PostgreSQL service should be used? This table outlines some of the advantages and disadvantages of each service, along with their PostgreSQL version support as of 4/2021.

| Service | Pros | Cons | Versions Supported |
|---|---|---|---|
| **Azure VM** | Any version, most flexible, fully PostgreSQL feature support | Customer responsible for updates, security, and administration | Any Version |
| **Single Server** | Minor version autoupgrades, no management overhead | Limited version support, no inbound logical replication support | 9.5 (deprecated), 9.6 (to be deprecated), 10, and 11 |
| **Flexible Server** | Minor version autoupgrades, no management overhead, support for replication with zone redundant HA, burstable SKUs, custom maintenance windows | Limited version support | 11, 12, and 13 |
| **Hyperscale Citus Server** | Minor version autoupgrades, no management overhead, Citus extensions installed | Limited version support, no inbound replication support | 11, 12, and 13 |

If running PostgreSQL 10 or lower and you do not plan to upgrade, the workload will need to run an Azure VM or Single Server. If the requirement is to target v13, utilize Flexible Server or Hyperscale Citus.

### Costs

After evaluating the entire WWI PostgreSQL data workloads, WWI determined they would need at least 4 vCores and 20GB of memory and at least 100GB of storage space with an IOP capacity of 450 IOPS. Because of the 450 IOPS requirement, they will need to allocate at least 150GB of storage due to Azure Database for PostgreSQL's IOPS allocation method. Additionally, they will require at least 7 days' worth of backups and one read replica. They do not anticipate an outbound egress of more than 5GB/month.

WWI intentionally chose to begin its Azure migration journey with a relatively small workload. However, the best practices of database migration still apply.

To determine these numbers, WWI installed Telegraf with the PostgreSQL Input Plugin to interface with the PostgreSQL statistics collector. Since the Plugin accesses the pg_stat_database and pg_stat_bgwriter tables, which show table and index access counts per-database and writer process statistics respectively, WWI has also made use of the following query to demonstrate the size of each user table in the reg_app schema. Note that the pg_table_size function solely includes table size, excluding the size of other associated objects, like indexes.

```
SELECT  s.table_name
     ,pg_size_pretty(s.Table_Size)
FROM
(
  SELECT  table_name
       ,pg_table_size('reg_app.' || table_name) AS Table_Size
  FROM information_schema.tables
  WHERE table_schema = 'reg_app'
  ORDER BY Table_Size DESC
) s;
```

Using the [Azure Database for PostgreSQL pricing calculator] WWI was able to determine the costs for the Azure Database for PostgreSQL instance. As of 4/2021, the total costs of ownership (TCO) is displayed in the following table for the WWI Conference Database:

| Resource | Description | Quantity | Cost |
|---|---|---|---|
| **Compute (General Purpose)** | 4 vcores, 20GB | 1 @ $0.351/hr | $3074.76 / yr |
| **Storage** | 5GB | 12 x 5 @ $0.115 | $6.90 / yr |
| **Backup** | 7 full backups (1x free) | 6 * 5(GB) * .10 | $3.00 / yr |
| **Read Replica** | 1 second region replica | compute + storage | $3081.66 / yr |
| **Network** | < 5GB/month egress | Free | |
| **Total** | | | $6166.32 / yr |

After reviewing the initial costs, WWI's CIO confirmed they will be on Azure for a period much longer than 3 years. They decided to use 3-year reserve instances to save an extra ~$4K/yr:

| Resource | Description | Quantity | Cost |
|---|---|---|---|
| **Compute (General Purpose)** | 4 vCores | 1 @ $0.1431/hr | $1253.56 / yr |
| **Storage** | 5GB | 12 x 5 @ $0.115 | $6.90 / yr |
| **Backup** | 7 full backups (1x free) | 6 * 5(GB) * .10 | $3.00 / yr |
| **Network** | < 5GB/month egress | Free | |
| **Read Replica** | 1 second region replica | compute + storage | $1260.46 / yr |
| **Total** | | | $2523.92 / yr (~39% savings) |

As the table above shows, backups, network egress, and any read replicas must be considered in the total cost of ownership (TCO). As more databases are added, the storage and network traffic generated would be the only extra cost-based factor to consider.

**Note:** The estimates above do not include any ExpressRoute, Azure App Gateway, Azure Load Balancer, or App Service costs for the application layers.

**Note**: The above pricing can change at any time and will vary based on region. The region used above was West US 2.

## Application Implications

When moving to Azure Database for PostgreSQL, the conversion to secure sockets layer (SSL) based communication is likely to be one of the biggest changes for the applications. SSL is enabled by default in Azure Database for PostgreSQL. It is likely the on-premises application and data workload is not set up to connect to PostgreSQL using SSL. When enabled, SSL usage will add some additional processing overhead and should be monitored.

**Warning**: Although SSL is enabled by default, it is possible to disable. This is strongly discouraged.

Follow the activities in Configure TLS connectivity in Azure Database for PostgreSQL - Single Server to reconfigure the application to support this strong authentication path.

Lastly, modify the server name in the application connection strings or switch the DNS to point to the new Azure Database for PostgreSQL server.

## Hyperscale Citus

When performing the assessment of source workloads, consider migrating from the basic PostgreSQL instance to a Hyperscale Citus instance. Some items that may affect the decision to move to Hyperscale include:

- Database Size has grown to 100s of GB of data

- The data workload is expected to grow by a factor of 10 or more
- Queries are slowing down
- The workload is approaching the hardware limits (CPU, Memory, Disk/IOPS) of a single node

There are also common workloads that are great fits for Hyperscale Citus:

- Real-time operational analytics (including time series)
- Multi-tenant SaaS applications
- IoT workloads (that need UPDATEs & JOINs)
- High-throughput transactional apps

Even though the target may meet the data points above, it is possible that changes must still be made to the database and application to support Hyperscale Citus.

- **Decide the sharding strategy** - Pick the distribution column/distribution key/sharding key that suits business needs.
- **Make changes to implement the sharding strategy** - This option could involve:
  - Updating some tables to add the sharding/distribution key
  - Deciding which tables to distribute
  - Determining which tables should become reference tables
  - Changing some foreign keys

**Note**: In most cases, the effort to change the schema is less work than implementing sharding in the application layer.

Explore some helpful links for evaluating an application and sharding strategy:

- Multi-tenant Applications
- Real-time Dashboards
- Timeseries Data

## WWI Use Case

WWI started the assessment by gathering information about their PostgreSQL data estate. As a result, they were able to compile the following:

| Name | Source | Size | IOPS | Version | Owner | Downtime |
|---|---|---|---|---|---|---|
| **WwwDB** | AWS (PaaS) | 1GB | 150 | 9.5 | Marketing Dept | 1 hr |
| **BlogDB** | AWS (Paas) | 1GB | 100 | 9.6 | Marketing Dept | 4 hrs |
| **ConferenceDB** | On-premises | 5GB | 50 | 9.5 | Sales Dept | 4 hrs |
| **CustomerDB** | On-premises | 10GB | 75 | 10.0 | Sales Dept | 2 hrs |
| **SalesDB** | On-premises | 20GB | 75 | 10.0 | Sales Dept | 1 hr |
| **DataWarehouse** | On-premises | 50GB | 200 | 10.0 | Marketing Dept | 4 hrs |

Each database owner was contacted to determine the acceptable downtime period. The planning and migration method selected was based on the acceptable database downtime.

For the first phase, WWI focused solely on the ConferenceDB database. The team needed the migration experience to assist in the proceeding data workload migrations. The ConferenceDB database was selected because of the simple database structure and the lenient downtime requirements. Once the database was migrated, the team focused on migrating the application into the secure Azure landing zone.

## Assessment Checklist

- Test the workload runs successfully on the target system.
- Ensure the right networking components are in place for the migration.
- Understand the data workload resource requirements.
- Estimate the total costs.
- Understand the downtime requirements.
- Prepare to make application changes.

# Planning

## Landing Zone

An Azure Landing zone is the target environment defined as the final resting place of a cloud migration project. In most projects, the landing zone should be scripted via ARM templates for its initial setup. Finally, it should be customized with PowerShell or the Azure Portal to fit the needs of the workload.

Since WWI is based in San Francisco, all resources for the Azure landing zone were created in the US West 2 region. The following resources were created to support the migration:

- Azure Database for PostgreSQL
- Azure Database Migration Service (DMS)
- Express Route
- Azure Virtual Network with hub and spoke design with corresponding virtual network peerings establish.
- App Service
- Application Gateway
- Private endpoints for the App Services and PostgreSQL instance

**Note:** As part of this guide, two ARM templates (one with private endpoints, one without) were provided to deploy a potential Azure landing zone for a PostgreSQL migration project. The private endpoints ARM template provides a more secure, production-ready scenario. Additional manual Azure landing zone configuration may be necessary, depending on the requirements.

## Networking

Getting data from the source system to Azure Database for PostgreSQL in a fast and optimal way is a vital component to consider in a migration project. Small unreliable connections may require administrators to restart the migration several times until a successful result is achieved. Restarting migrations due to network issues can lead to wasted effort, time, and money.

Take the time to understand and evaluate the network connectivity between the source, tool, and destination environments. In some cases, it may be appropriate to upgrade the internet connectivity or configure an ExpressRoute connection from the on-premises environment to Azure. Once on-premises to Azure connectivity has been created, the next step is to validate that the selected migration tool can connect from the source to the destination.

The migration tool location will determine the network connectivity requirements. As shown in the table below, the selected migration tool must connect to both the on-premises machine and Azure. Azure should be configured to accept network traffic from only the migration tool location. Notice the two main ports 5432 and 6432:

| Migration Tool | Type | Tool Location | Inbound Network Requirements | Outbound Network Requirements |
|---|---|---|---|---|
| **Azure Data Factory (ADF)** | Offline | Azure or Hybrid | Allow 5432 (6432 with PgBouncer) from external IP | A path to connect to the Azure PostgreSQL database instance |
| **Database Migration Service (DMS)** | Online or Offline | Azure | Allow 5432 (6432 with PgBouncer) from external IP | A path to connect to the Azure PostgreSQL database instance |
| **Import/Export (pgAdmin, pg_dump)** | Offline | On-premises | Allow 5432 (6432 with PgBouncer) from internal IP | A path to connect to the Azure PostgreSQL database instance |
| **Import/Export (pgAdmin, pg_dump)** | Offline | Azure VM | Allow 5432 (6432 with PgBouncer) from external IP | A path to connect to the Azure PostgreSQL database instance |
| **pg_dump/pg_dumpall** | Offline | On-premises | Allow 5432 (6432 with PgBouncer) from internal IP | A path to connect to the Azure PostgreSQL database instance |
| **pg_dump/pg_dumpall** | Offline | Azure VM | Allow 5432 (6432 with PgBouncer) from external IP | A path to connect to the Azure PostgreSQL database instance |
| **Logical replication (10.0+, only available for Flexible Server migrations)** | Online | On-premises | Allow 5432 (6432 with PgBouncer)from external IP or private IP via Private endpoints | A path for each replication server to the master |
| **Slony/Londiste** | Online | On-premises | Allow 5432 (6432 with PgBouncer) from external IP | A path to connect to the Azure PostgreSQL database instance |
| **Log shipping** | Online or Offline | Azure VM | Allow 5432 (6432 with PgBouncer) from external IP | A path to connect to the Azure PostgreSQL database instance |

We will discuss these migration methods in more detail in the next section.

**Note**: pgAdmin's **Backup and Restore** feature uses the pg_dump, pg_dumpall, and pg_restore utilities. They will be discussed in much more detail later on, but pg_dumpall is used to backup all databases in the PostgreSQL server instance, and global objects, like database users.

Other networking considerations include:

- DMS located in a VNET will be assigned a dynamic public IP to the service. At creation time, place the service inside a virtual network that has connectivity via ExpressRoute or over a site-to-site VPN.
- DMS can be configured in a hybrid model with a worker installed on-premises to proxy the data to DMS.
- When using an Azure Virtual Machine to run the migration tools, assign it a public IP address and then only allow it to connect to the on-premises PostgreSQL instance.
- Outbound firewalls must ensure outbound connectivity to Azure Database for PostgreSQL. The PostgreSQL gateway IP addresses are available on the Connectivity Architecture in Azure Database for PostgreSQL page.

## Private Link and/or VNet integration

As noted above, not all Azure Database for PostgreSQL services support private links.

| Service | Support |
|---|---|
| **Single Server** | Only General Purpose and Memory Optimized pricing tiers will allow for private link configurations. |
| **Flexible Server** | Supports VNet integration with limitations. Service in preview. |
| **Hyperscale Citus** | Does not support Private Link and/or VNet integration; please use firewall rules for network security. |

## Azure Arc

Azure Arc is a software solution that enables the surfacing of on-premises and multi-cloud resources, such as virtual or physical servers and Kubernetes clusters, into Azure Resource Manager (ARM). This enables the **management of non-Azure resources as if they are running in Azure**, using a single pane of glass to manage the entire data estate. PostgreSQL Hyperscale Citus can be deployed on customer infrastructure with Azure Arc enabled and be visible in the Azure Portal for management purposes.

## SSL/TLS Connectivity

In addition to the application implications of migrating to SSL-based communication, the SSL/TLS connection types are also something that needs to be considered. After creating the Azure Database for PostgreSQL database, review the SSL settings, and read the SSL/TLS connectivity in Azure Database for PostgreSQL article to understand how the TLS settings can affect the security posture.

## WWI Use Case

WWI's cloud team has created the necessary Azure landing zone resources in a specific resource group for the Azure Database for PostgreSQL. Additional resources will be included to support the applications. To create the landing zone, WWI decided to script the setup and deployment using ARM templates. ARM templates make it possible to quickly tear down and re-create the environment.

As part of the ARM template, all connections between virtual networks will be configured with peering in a hub and spoke architecture. The database and application will be placed into separate virtual networks. An Azure App Gateway will be placed in front of the app service to allow the app service to be isolated from the Internet. The Azure App Service will connect to the Azure Database for PostgreSQL using a private endpoint.

WWI originally wanted to test an online migration, but the required network setup for DMS to connect to their on-premises environment made this infeasible. WWI chose to do an offline migration instead. The PostgreSQL pgAdmin tool was used to export the on-premises data and then was used to import the data into the Azure Database for PostgreSQL instance. The WWI migration team has also learned the versatile Azure Data Studio tool has preview PostgreSQL support, and would like to explore its utility for developing applications using PostgreSQL.

## Planning Checklist

- Prepare the Azure landing zone. Consider using ARM template deployment in case the environment must be torn down and rebuilt quickly.
- Verify the networking setup. Verification should include testing connectivity, bandwidth, latency, and firewall configurations.
- Determine if you are going to use the online or offline data migration strategy.
- Decide on the SSL certificate strategy.

# Migration Methods

Migrating data from the source to the target will require PostgreSQL tools and features.

Complete the entire assessment and planning stages before starting the next stages. The decisions and data collected are migration path and tool selection dependencies.

We explore the following commonly used tools in this section:

- PostgreSQL pgAdmin
- pg_dump/pg_dumpall
- COPY command
- Azure Data Factory (ADF)
- 3rd party tools
- Replication
- Azure Database Migration Service (DMS)

## PostgreSQL pgAdmin

PostgreSQL pgAdmin provides a rich GUI experience that allows developers and administrators to design, develop, and manage their PostgreSQL instances.

The latest versions of PostgreSQL pgAdmin provide sophisticated object migration capabilities when moving a database and global objects from a source to a target.

### Data Import and Export

#### pgAdmin

PostgreSQL pgAdmin provides a wizard-based UI to do full or partial export and import of tables. The export and import process supports three file targets: .bin, .csv and .txt.

#### Azure Data Studio

A tool similar to the commonly used PostgreSQL pgAdmin tool is Azure Data Studio. Download Azure Data Studio and the corresponding PostgreSQL extension to manage PostgreSQL databases.

#### Visual Studio Code

Visual Studio Code also has PostgreSQL extensions available for connecting to PostgreSQL and performing various tasks such as connect to PostgreSQL instance, manage connection profiles, write and run queries, etc.

#### Backup and Restore

Use pgAdmin to call the pg_dump, pg_dumpall, and pg_restore commands via UI dialogs.

## Dump and restore (pg_dump/pg_dumpall)

pg_dump and pg_dumpall are provided as part of the PostgreSQL installation. These client utilities can create logical backups via SQL scripts. The SQL statements can be replayed to rebuild the database to a point in time. The pg_dump utility is not intended to be a fast or a scalable solution for backing up or migrating very large amounts of data. Executing a large set of SQL insert statements can perform poorly due to the disk I/O required to update indexes. However, when combined with other tools that require the original schema, pg_dump is a great tool for generating the database and table schemas. The schemas can create the target landing zone environment.

### PostgreSQL COPY Command

PostgreSQL comes out of the box with the COPY command. This command can run in two variants, server-based and client-based. The server-based version is not allowed to run on Azure Database for PostgreSQL. Therefore, migrations require using the client-based version \COPY.

**Note**: The COPY command cannot be used on Azure Database for PostgreSQL due to a lack of superuser permissions.

Some things to note about the \COPY command:

- \COPY command uses pipes, so no space required on the client-side. This means that \COPY can also obtain data from programs like wget
- Potential for increased performance with parallelism, compared to pg_dump
- \COPY method does not migrate indexes, foreign keys, and other similar dependency objects. Additional steps are needed to move those objects.

**Note**: Internally, pg_dump and pg_restore use the COPY command.

When using the COPY command, it is possible to implement both a non-parallel approach (such as pg_dump) or a parallel approach (COPY using pipes and scripting).

## Azure Data Factory (ADF)

Similar to pgAdmin import and export, it is possible to export table data from a source to a target using Azure Data Factory (ADF). ADF trigger mechanisms build and use pipelines to move data, either manually or based on a schedule. See Copy data from PostgreSQL by using Azure Data Factory for more information.

## Other open-source tools

PostgreSQL workloads can be migrated easily using 3rd party migration tools. The time savings and ease of use may come with a price. The tools may add extra time costs to the migration.

Some of these include:

- Slony-I

- Londiste
- Bucardo

PostgreSQL 9.0+ includes streaming replication, which, for many use cases, is likely to be simpler and more convenient than 3rd party tools. However, 3rd party tools can provide additional levels of granularity and configuration over the replication process in cases such as:

- Migrating between PostgreSQL versions. 3rd party tools can cope with having nodes running different versions of PostgreSQL.
- Write-ahead log (WAL)-based replication requires that all databases use identical versions of PostgreSQL, running on identical architectures.
- Replicate parts of the changes that are going on. WAL-based replication duplicates absolutely everything.
- Require extra behaviors to run on subscribers, such as populating cache management information.
- WAL-based replication duplicates absolutely everything, and nothing extra that changes data can run on a WAL-based replica.

Have questions about 3rd party tools? Feel free to reach out to the AskAzureDBforPostgreSQL@service.microsoft.com.

## Replication

Similar to other database management systems, PostgreSQL provides several ways to replicate data to another PostgreSQL instance. These include:

- Physical Replication/Log Shipping/Warm Standby.
- Logical Replication.
- Logical Decoding.

**Note**: For a complete list of replication solutions (including those not supported via any directional flows in Azure Database for PostgreSQL), reference High Availability, Load Balancing, and Replication - Comparison of Different Solutions in the PostgreSQL documentation pages.

### Synchronous vs. Asynchronous

As with other replication technologies in other database management systems, there are several supported options for sending the transaction data to the targets. In synchronous replication, the source doesn't finish committing until a replica confirms it received the transaction. In asynchronous streaming replication, the replica(s) can fall behind the source when the source is faster/busier. If the source crashes, it is possible that some data wasn't replicated.

### Logical Replication

Each change is sent to one or more replica servers directly over a TCP/IP connection as it happens. The replicas must have a direct network connection to the master configured in their recovery.conf's primary_conninfo option.

To use the logical replication feature, there are some setup requirements:

- Database source must be 9.4 or higher and the target must be the same or higher version.
- Tables must have a primary key or changes may not get synced to the target
- A user on the target system must be a superuser
- Migration users must have permissions to configure logging and create new users on the master server.
- Ensure that the target machine can gain access to the master server (firewalls, IP address, etc.).

**Note**: Logical replication is not a viable option for moving to Azure Database for PostgreSQL Single Server. However, Flexible Server using PostgreSQL V11 or higher does support native logical replication.

### Logical Decoding

Logical decoding works by exposing the WAL log files in a way they can be consumed by an external application (such as Debezium). The log changes can be replayed on the target environment. In most cases, there are no direct links between the source and the targets, and the external application acts as a proxy to send changes to the targets.

Logical decoding works by creating slots on the source, which capture WAL changes. The results are returned in a static format (such as wal2json) that the external tool must be able to read. If the external application is not reading the data from the slot, the unconsumed logs will pile up and eventually fill up the local storage.

For more in-depth information about Logical Decoding, reference this post.

### Trigger-based

Rather than mapping and exposing WAL log files, it is also possible to implement triggers to send the updates to the target standby servers. An example of this type of replication pattern is implemented by the Slony-I product. This path is preferable when using older versions of PostgreSQL (9.4 or lower) where Logical Replication and Logical Decoding features are not supported.

**Supported replication paths**

| Replication Type | Target Service | Direction | Supported | Version Support | Notes |
|---|---|---|---|---|---|
| **Physical/File system/Block Device Replication** | Single Server, Flexible Server, Hyperscale Citus | Ingress/Egress To Azure | Not Supported | 9.0 or higher | Requires file system access |
| **WAL Log Shipping** | Single Server, Flexible Server, Hyperscale Citus | Ingress/Egress To Azure | Not Supported | N/A | Requires file system access |
| **Logical Replication** | Single Server | Ingress To Azure | Not Supported | N/A | Requires superuser |
| **Logical Decoding** | Single Server, Flexible Server, Hyperscale Citus | Ingress/Egress To Azure | Supported | 9.6 or higher | N/A |
| **Trigger-based** | Single Server, Flexible Server, Hyperscale Citus | Ingress/Egress To Azure | Supported | Any | 3rd Party tool required |

## Azure Database Migration Service (DMS)

The Azure Database Migration Services (DMS) is an Azure cloud-based tool that allows administrators to keep track of the various settings for migration and reuse them if necessary. DMS works by creating migration projects with settings that point to various sources and destinations. It supports both online and offline migrations, with the online option allowing for significantly minimized downtime. Additionally, it supports on-premises data workloads and cloud-based workloads such as Amazon Relational Database Service (RDS) for PostgreSQL.

Although the DMS service is an online tool, it does rely on the logical replication feature of PostgreSQL to complete its tasks. Currently, DMS partially automates the offline migration process. DMS requires the generation and application of the matching schema in the target Azure Database for PostgreSQL instance. Schemas can be exported using the pg_dump client utility.

Review the Known issues/migration limitations with online migrations from PostgreSQL to Azure DB for PostgreSQL for specific details on configuration and prerequisites for DMS to properly migrate database(s).

DMS is recommended for data migrations up to 1TB.

## Fastest/Minimum Downtime Migration

There are plenty of paths for migrating the data. Deciding which path to take is a function of the migration team's skill set, and the amount of acceptable database and application downtime. Some tools support multi-threaded parallel data migration approaches. Other tools were designed for simple migrations of table data only.

The fastest and most complete path is to use replication-style features (either directly with PostgreSQL, DMS, or 3rd party tools), but replication typically comes with the costs of adding primary keys, which could break the application and force costly coding changes.

## Decision Table

There are many paths WWI can take to migrate their PostgreSQL workloads. We have provided a table of the potential paths and the advantages and disadvantages of each:

| Objective | Description | Tool | Prerequisites | Advantages | Disadvantages |
|---|---|---|---|---|---|
| **Fastest migration possible** | Parallel approach | pg_dump/pg_dumpall | Scripted Setup | Highly parallelized | Target throttling |
| **Fastest migration possible** | Parallel approach | Azure Data Factory | ADF Resource, Linked Services setup, Pipelines | Highly parallelized | Target throttling |
| **Online migration** | Keep the source up for as long as possible | Logical replication | None | Seamless | Extra processing and storage |
| **Online migration** | Keep the source up for as long as possible | Logical decoding | 3rd party tools | High-performance, zero-downtime, high availability, support for other targets | Extra setup, processing and storage |
| **Online migration** | Keep the source up for as long as possible | Trigger-based replication | 3rd party tool | Seamless | 3rd party tool configuration |
| **Online migration** | Keep the source up for as long as possible | Azure Database Migration Service (DMS) | None | Repeatable process | Limited to data only, supports 10.0 and higher |
| **Highly Customized Offline Migration** | Selectively export objects | pg_dump | None | Highly customizable | Manual |

| | | | | | |
|---|---|---|---|---|---|
| **Offline Migration Semi-automated** | UI based export and import | PostgreSQL pgAdmin | Download and Install | Semi-automated | Only common sets of switches are supported |

## WWI Use Case

WWI has selected its Conference database as its first migration workload. The Conference database selected because it had the least risk and the most available downtime due to the gap in the annual conference schedule. The database does not use unsupported Azure Database for PostgreSQL services in the target environment. The migration team will attempt to perform an offline migration using the pg_dump/pg_restore PostgreSQL tools.

During their assessment period, they did find the customer database does use some languages, extensions, and a custom function not available in the target service for the conference database. The migration team asked the development team to replace those features while they migrate the simpler workloads. If the migration problems can be solved successfully, the migration team will choose an Azure Database for PostgreSQL service. Otherwise, they will provision an Azure VM to host the workload.

## Migration Methods Checklist

- Ensure the correct migration method is selected given the target and source environments.
- Ensure the migration method can meet the business requirements.
- Verify the data workload will support the method.

# Test Plans

## Overview

WWI created a test plan that included a set of IT and Business tasks. Successful migrations require all the tests to be executed.

Tests:

- Ensure the migrated database has consistency (same record counts and query results) with on-premises tables.
- Ensure the performance is acceptable (it should match the same performance as if it were running on-premises).
- Ensure the performance of target queries meets stated requirements.
- Ensure acceptable network connectivity between on-premises and the Azure network.
- Ensure all identified applications and users can connect to the migrated data instance.

WWI identified a migration weekend and time window that started at 10 PM and ended at 2 AM Pacific Time. If the migration did not complete before the 2 AM target (the 4hr downtime target) with all tests passing, the rollback procedures were started. Issues were documented for future migration attempts. All migrations windows were pushed forward and rescheduled based on acceptable business timelines.

## Sample Queries

A series of queries were executed on the source and target to verify migration success. The following queries and scripts will help determine if the migration moved all required database objects from the source to the target.

### Table rows

Use this query to get all the tables and an estimated row count. Replace the {SCHEMA_NAME} (the sample uses reg_app):

```
select table_schema,
      table_name,
      (xpath('/row/cnt/text()', xml_count))[1]::text::int as row_count
from (
  select table_name, table_schema,
        query_to_xml(format('select count(*) as cnt from %I.%I', table_schema, table_name), false, true, '') as xml_cou
nt
  from information_schema.tables
  where table_schema = '{SCHEMA_NAME}'
) t
```

### Table Fragmentation

The data tables are likely to continue to grow larger with continued application use. The data may not grow in some cases, but it is constantly changing through deletions and updates. Database changes cause fragmentation in the database files. The PostgreSQL REINDEX statement can reduce physical storage space needs and improve I/O efficiency. The space reduction is performed via the autovacuum features. Also, it can be executed manually as well.

Optimize the PostgreSQL table indexes by running the following:

```
REINDEX TABLE {SCHEMA_NAME}.{TABLE_NAME}
```

It can also run at the database scope:

```
REINDEX DATABASE {DB_NAME}
```

Reference Index Maintenance for more information on performing routine re-indexing in PostgreSQL.

### Database Bloat

When determining the target server size, review the database bloat first. This can help give a better idea of the actual storage consumption after the migration. Reference the Show database bloat article on the PostgreSQL wiki.

### Database objects

Use the following queries to ensure all database objects have been migrated.

**Note**: These queries can account for the counts, they may not account for the version of the particular database object. For example, even though a stored procedure may exist, it could have changed between the start and end of the migration.

**Users**

```
SELECT usename AS role_name,
  CASE
    WHEN usesuper AND usecreatedb THEN
    CAST('superuser, create database' AS pg_catalog.text)
    WHEN usesuper THEN
      CAST('superuser' AS pg_catalog.text)
    WHEN usecreatedb THEN
      CAST('create database' AS pg_catalog.text)
    ELSE
      CAST('' AS pg_catalog.text)
  END role_attributes
FROM pg_catalog.pg_user
ORDER BY role_name desc;
```

**Indexes**

```
select *
from pg_indexes
where tablename not like 'pg%';
```

**Constraints (Foreign and Primary Keys)**

```
SELECT con.*
FROM pg_catalog.pg_constraint con
INNER JOIN pg_catalog.pg_class rel
        ON rel.oid = con.conrelid
INNER JOIN pg_catalog.pg_namespace nsp
        ON nsp.oid = connamespace
```

**Tables**

```
select table_name, table_schema
from information_schema.tables
where table_type = 'BASE TABLE'
and table_schema = schema_name
order by table_name;
```

**Views**

```
select *
from INFORMATION_SCHEMA.views
where table_schema = '{SCHEMA_NAME}'
```

**Stored Procedures (v11+)**

```
SELECT  *
FROM    pg_catalog.pg_namespace n
JOIN    pg_catalog.pg_proc p
ON      pronamespace = n.oid
WHERE   nspname = 'public';
```

**Functions**

```
select n.nspname as function_schema,
       p.proname as function_name,
       l.lanname as function_language,
       case when l.lanname = 'internal' then p.prosrc
            else pg_get_functiondef(p.oid)
            end as definition,
       pg_get_function_arguments(p.oid) as function_arguments,
       t.typname as return_type
from pg_proc p
left join pg_namespace n on p.pronamespace = n.oid
left join pg_language l on p.prolang = l.oid
left join pg_type t on t.oid = p.prorettype
where n.nspname not in ('pg_catalog', 'information_schema')
order by function_schema,
         function_name;
```

**Triggers and Event Triggers**

```
select event_object_schema as table_schema,
       event_object_table as table_name,
       trigger_schema,
       trigger_name,
       string_agg(event_manipulation, ',') as event,
       action_timing as activation,
```

```
    action_condition as condition,
    action_statement as definition
from information_schema.triggers
group by 1,2,3,4,6,7,8
order by table_schema,
      table_name;
```

**Casts**

```
select *
from pg_cast pc, pg_proc pp
where pc.castfunc = pp.oid
and pronamespace != 11
```

**Languages**

```
SELECT *
FROM pg_language
```

**Extensions**

```
SELECT *
FROM pg_extension
```

**Foreign Data Wrappers and Foreign Tables**

```
SELECT fdw.fdwname AS "Name",
  pg_catalog.pg_get_userbyid(fdw.fdwowner) AS "Owner",
  fdw.fdwhandler::pg_catalog.regproc AS "Handler",
  fdw.fdwvalidator::pg_catalog.regproc AS "Validator"
FROM pg_catalog.pg_foreign_data_wrapper fdw
ORDER BY 1;
```

**Note**: The postgre_fdw extension is available in Azure Database for PostgreSQL. However, the file_fdw extension and the ability to add new FDW handlers are not supported. Look for any FDWs that utilize anything but the postgre_fdw extension.

**User-Defined Types**

```
SELECT     n.nspname as schema, t.typname as type
FROM       pg_type t
LEFT JOIN   pg_catalog.pg_namespace n ON n.oid = t.typnamespace
WHERE      (t.typrelid = 0 OR (SELECT c.relkind = 'c' FROM pg_catalog.pg_class c WHERE c.oid = t.typrelid))
AND     NOT EXISTS(SELECT 1 FROM pg_catalog.pg_type el WHERE el.oid = t.typelem AND el.typarray = t.oid)
AND     n.nspname NOT IN ('pg_catalog', 'information_schema');
```

**Note**: It is also possible to perform any of the above queries using the psql tool. To display the SQL queries the psql tool sends to PostgreSQL, add the --echo-hidden parameter.

### Dependency Walker

When inheriting a PostgreSQL environment, understanding how to re-install items that require 3rd party software will be important when moving to an Azure Virtual Machine environment. Use the Dependency Walker tool to find any required executables and dependent libraries. For example, the Perl programming language support for custom functions and stored procedures is a custom installation.

## Rollback Strategies

The above queries will provide a list of object names and counts used in a rollback decision. Migration users can take the first object verification step by checking the source and target object counts. A failure in counts may not necessarily mean that a rollback is needed. Performing an in-depth evaluation could point out that the discrepancy is small and easily recoverable. Manual migration of failed objects may be possible.

For example, if all tables and rows were migrated, but only a few of the functions were missed, remediate those failed objects and finalize the migration. If the database is relatively small, it could be possible to clear the Azure Database for PostgreSQL and restart the migration again. Small database migrations can be restarted if problems have been remediated.

Large database migrations may need to stopped and rolled back due to the time required to restart.

Identifying missing database objects needs to occur quickly during a migration window. Every minute counts. One option could be exporting the environment object names to a file and using a data comparison tool to reduce the missing object verification time. Another option could be exporting the source database object names and importing the data into a target database environment temp table. Compare the data using a **scripted** and **tested** SQL statement. Data verification speed and accuracy are critical to the migration process. Do not rely on reading and verifying a long list of database objects during a migration window. Manage by exception.

### Decision Table

| Discrepancy | Time To Sync | Rollback? | Resolution Path |
|---|---|---|---|
| **One Table (Row mismatch)** | Less than the remaining window | No | Sync the mismatched table |
| **One Table (Row mismatch)** | More than the remaining window | Yes | Start the Rollback |
| **Several Tables (Row mismatch)** | More than the remaining window | Yes | Start Rollback |
| **A few functions** | Less than the remaining window | No | Sync the missing functions |

In the data-migration section, we will provide a database migration inventory script that will provide object counts that can be used to compare source and destination after a migration path has been completed.

## WWI Use Case

The WWI CIO received a confirmation report that all database objects were migrated from the on-premises database to the Azure Database for PostgreSQL instance. The database team ran the above queries against the database before the beginning of the migration and saved all the results to a spreadsheet.

The source database schema information was used to verify the target migration object fidelity.

## Checklist

- Have test queries scripted, tested, and ready to execute. Manual comparisons should be avoided.
- Know how long test queries take to run and make them a part of the migration timeline.
- Have a mitigation and rollback strategy ready for different potential outcomes.
- Have a well-defined timeline of events for the migration.

# Performance Baselines

Understanding the existing PostgreSQL workload is one of the best investments to ensure a successful migration. Excellent system performance depends on adequate hardware and great application design. Items such as CPU, memory, disk, and networking need to be sized and configured appropriately for the anticipated load. Hardware and configuration are part of the system performance equation. The developer must understand the database query load and the most expensive queries to execute. Focusing on the most expensive queries can make a big difference in the overall performance metrics.

At the beginning of the project, create query performance baselines. The performance baselines should be used to verify the Azure landing zone configuration for the migrated data workloads. Most systems will be run 24/7 and have different peak load times. Capture the peak workloads for the baseline. In a later document section, we will explore the source server parameters and how they are essential to the overall performance baseline picture. The server parameters must be elevated during a migration project.

## Tools

Below are tools used to gather server metrics and the database workload information. Use the captured metrics to determine the appropriate Azure Database for PostgreSQL tier and the associated scaling options.

- pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands repeatedly, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction.
- PostgreSQL Enterprise Monitor requires a paid license. The enterprise edition tool can provide a sorted list of the most expensive queries, server metrics, file I/O and topology information.
- Percona Monitoring and Management (PMM) is a best-of-breed open source database monitoring solution. It helps to reduce complexity, optimize performance, and improve the security of business-critical database environments, no matter the deployed location.

## Server Parameters

PostgreSQL runtime functionality is driven by several server parameters. These parameters are mainly stored in the postgresql.conf file, but they can be broken into sub configurations by using include and include_dir statements in the main configuration file. This configuration applies to on-premises environments. When dealing with Azure Database for PostgreSQL instances, you will not have access to these files directly, but rather through the Azure Portal and API calls with PowerShell or Azure CLI.

**Note**: When evaluating the current server parameters for migration, thoroughly review include and include_dir statements and bring over any externally referenced parameters.

PostgreSQL server default configurations may not adequately support a workload. There are many server parameters in PostgreSQL. In most cases, the migration team should focus on a small handful. The following parameters should be evaluated in the **source** and **target** environments. Incorrect configurations can affect the speed of the migration. We will revisit these parameters when we execute the migration steps in later sections.

- **listen_addresses**: By default, PostgreSQL only responds to connections from localhost. If the server needs to be accessible from other systems via standard TCP/IP networking, modify listen_addresses from its default.
- **wal_buffers**: PostgreSQL writes its WAL (write-ahead log) record into the buffers, and then these buffers are flushed to disk. The default setting is 16MB, but when there are many concurrent connections, a higher value can give better performance.
- **checkpoint_timeout**: Set time between WAL checkpoints. A setting that is too low decreases crash recovery time, but it hurts performance too since every checkpoint ends up consuming valuable system resources.
- **min_wal_size** and **max_wal_size**: Sets the size boundaries on the transaction log of PostgreSQL. This parameter is the amount of data that can be written until a checkpoint is issued, which in turn syncs the in-memory data with the on-disk data.
- **max_connections**: The default is 100 connections. Using a connection pool at the application level is preferred, but the server connection configuration may need to increase as well. We will explore **PgBouncer** and its connection pooling feature in the data-migration section. PostgreSQL forks a new process for every new connection. By using a connection pool, connections are initiated at startup. This feature reduces connection latency for applications. PgBouncer can run on VMs or even Azure Kubernetes Service.
- **shared_buffers**: Governs the amount of memory used by PostgreSQL for caching data and has a default of 128MB. Reasonable potential values for this parameter would be around one-fourth of the system memory.
- **temp_buffers**: Defines the amount of memory used by each database session for caching temporary table data.
- **maintenance_work_mem**: Defines the maximum amount of memory used by maintenance operations (ex: VACUUM, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY).
- **max_worker_processes**: This is the number of background processes the database engine can use. Setting this parameter will require a server restart. The default is 8.
- **work_mem**: If there are many complex sorts which require significant memory, then increasing this parameter allows PostgreSQL to do larger in-memory sorts.

- **effective_io_concurrency**: The number of real concurrent IO operations supported by the IO subsystem. As a starting point, with plain HDDs, try setting at 2; with SSDs, go for 200; and if there is a large SAN backing the database, start with 300.
- **effective_cache_size**: This parameter is used by the PostgreSQL query planner to calculate the plans the planner expects to fit in RAM. Sets the planner's assumption about the effective size of the disk cache that is available for a single query.

Run the following command to export all server parameters to a file for review. Using the exported values, create a script to reapply the Azure Database for PostgreSQL server parameters after the migration.

Reference Configure server parameters in Azure Database for PostgreSQL using the Azure portal.

```
select *
from pg_file_settings
```

**Note**: The SHOW command can be used to show the value of a specific server parameter.

Before migration begins, export the source PostgreSQL configuration settings. Compare those values to the Azure landing zone instance settings after the migration. Modified Azure landing zone settings should be reset after the migration. Also, the migration user should verify the server parameters can be set before the migration.

**Note**: For a simple but effective tool for finding server parameter values, reference PGTune.

### Egress and Ingress

Modify the source and the target PostgreSQL server parameters for the fastest possible egress and ingress. The parameters could be different for each migration tool and path. For example, a tool that performs a migration in parallel may need more connections on the source and the target versus a single-threaded tool.

Review any timeout parameters that may be affected by the datasets:

**statement_timeout**: By default, this parameter is turned off. Check that it has not been enabled when dumping large database tables.

Additionally, review any parameters that will affect maximums:

**work_mem**: When performing sorts on database exports based on a particular key or column, that sort operation will be performed in memory. Be sure this setting is set to an amount that will ensure a fast sort of results before export.

## WWI Use Case

WWI reviewed its Conference database workload and determined it had a very small load. Although a basic tier server would work for them, they did not want to perform work later to migrate to another tier. The server being deployed will eventually host the other PostgreSQL data workloads, so they picked the General Purpose tier.

The PostgreSQL 9.5 server is running with the default server parameters set during the initial install.

# Data Migration with Backup and Restore

## Setup

Follow all the steps in the Setup guide to create an environment to support the following steps.

## Configure Server Parameters

To support the migration, set the source and target PostgreSQL instance parameters to allow for faster egress and ingress. Follow the steps in data-migration-1.

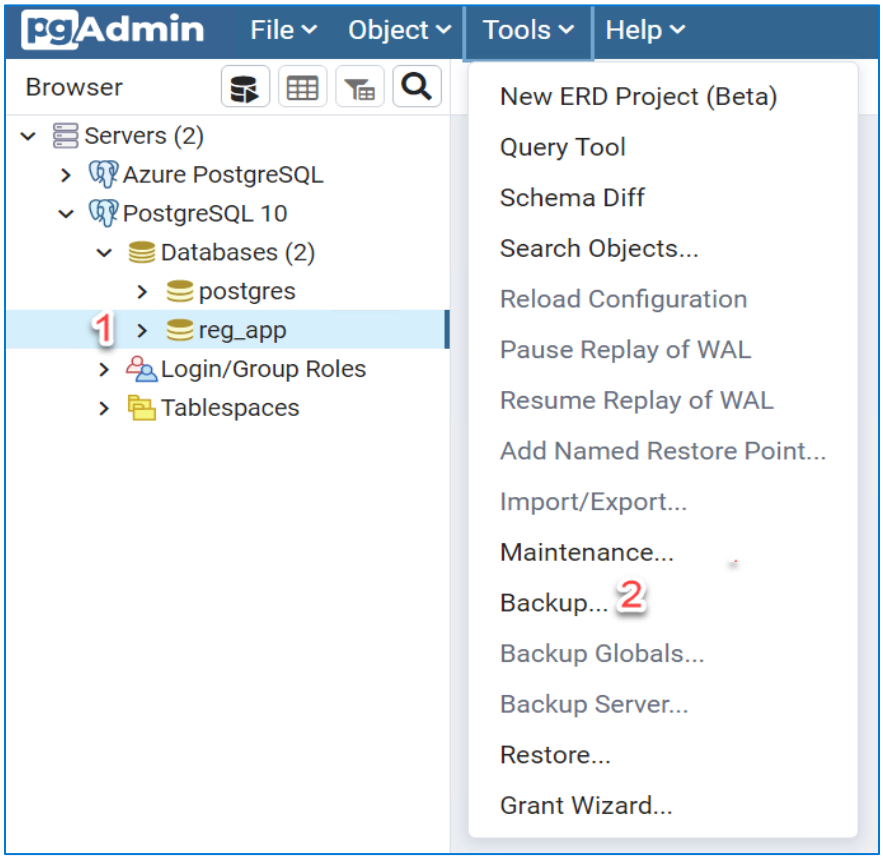
## Data

### Manual Backup

- Open PostgreSQL **pgAdmin** and connect as the local database's postgres user.

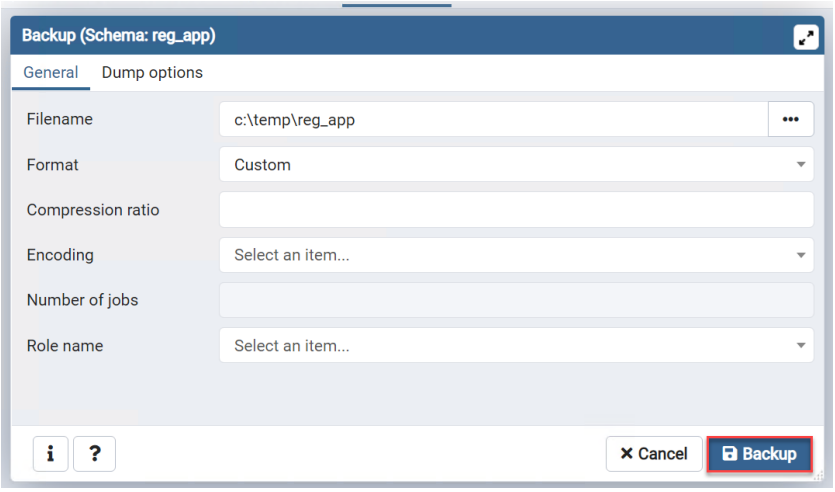
**Note:** Although the steps here use pgAdmin, it is possible to use the Azure Data Studio tool to make a database backup. This tool will call pg_dump the same as pgAdmin. Similarly, it is possible to perform several management tasks using the Visual Studio Code PostgreSQL extension.

- Expand the **Databases** node
- Select the **reg_app** database.
- In the application menu, select **Tools->Backup**



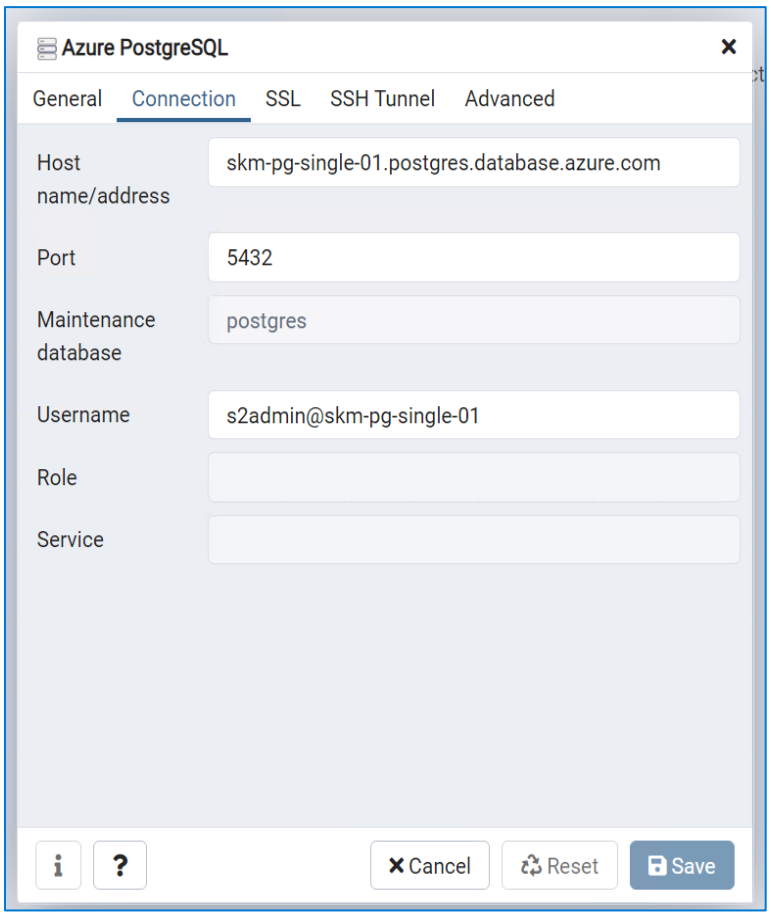

Backing up the local reg_app database using the pgAdmin tool.

- For the file name, type C:\temp\reg_app).
- For the format, select **Custom**
- Select **Backup**



Setting the destination path and file type for the generated backup file.

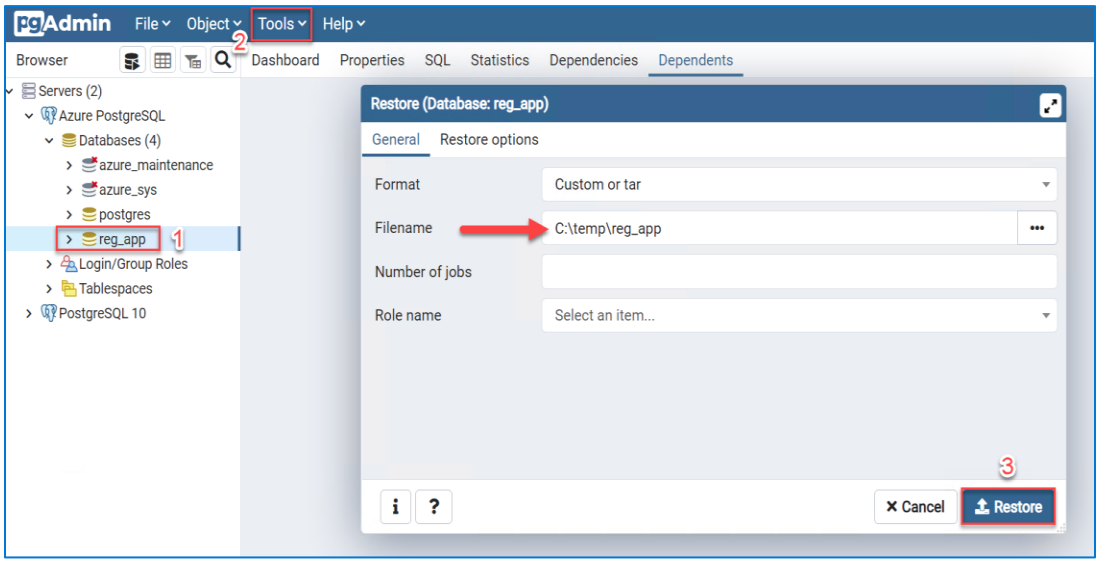
- In PostgreSQL pgAdmin, create a new connection to the Azure Database for PostgreSQL instance.
  - For **Name**, enter a name such as Azure PostgreSQL
  - Select the **Connection** tab
  - For **Hostname**, enter the full server DNS (ex: servername.PostgreSQL.database.azure.com).
  - Enter the **username** (ex: s2admin@servername). Remember that the username, without the servername, is acceptable for Azure Database for PostgreSQL Flexible Server.
  - Enter the **password** (**Seattle123Seattle123**)
  - Select the **SSL** tab.
  - For the **Root certificate**, browse to the **c:\temp\BaltimoreCyberTrustRoot.crt.pem** key file.
  - Select **Save**.



PostgreSQL connection dialog box is displayed.

### Manual Restore (Attempt #1)

- For the Azure Database instance, expand the **Databases** node
- Right-click the **Databases** node and select **Create->Database**
- For the **name**, type reg_app
- Select **Save**
- Select the **reg_app** database.
- In the menu, select **Tools->Restore**




Using the backup file to recreate the database on the Azure PostgreSQL instance.

- Browse to the c:\temp\reg_app file
- Select **Restore**
- Expect to see many errors. In the toast notification, click **More details**. Review the errors. Notice some relate to roles, others relate to unsupported items, and many relate to the requirement for the superuser permission.

**Note:** Doing a full backup will also include items that may not be supported in the target environment. In this case, the several items will need to be removed before it is possible to use the backup to support the items that need to be migrated.

The same pg_restore command will look like the following:

```
pg_restore.exe --host "servername.postgres.database.azure.com" --port "5432" --username "s2admin@servernam
e" --no-password --dbname "reg_app" --verbose "c:\\temp\\reg_app"
```

### Manual Backup (Attempt #2)

- For the local instance, expand the **Databases** node
- Select the **reg_app** database.
- In the application menu, select **Tools->Backup**
- For the file name, type C:\temp\reg_app.sql).
- For the format, select **Plain**
- Select **Backup**

### Manual Restore (Attempt #2)

- For the Azure Database instance, expand the **Databases** node
- Right-click the database, select **Query Tool**
- Open the c:\temp\reg_app.sql file
- Press **F5** to run the file. Observe the syntax errors.

**Note**: It is not possible to run the COPY command in the pgAdmin tool. Use the psql command line for this purpose.

### Manual Restore (Attempt #3)

- Use psql to execute the script:

```
psql -h servername.postgres.database.azure.com -p 5432 -d reg_app -U s2admin@servername -f "c:\temp\reg_ap
p.sql"
```

- Notice several errors are displayed. The script contains several unsupported features from the database export.

#### Removing unsupported features

- Remove the unsupported features until the script runs successfully. Some hints:
- Export and add all source users (reference the schema migration section for exporting users and roles)
- Replace all CREATE statements with corresponding CREATE IF NOT EXISTS statements and CREATE OR REPLACE statements
- Update all postgres owners with s2admin or conferenceadmin
- Remove the addition of procedural languages (plperl)

- Remove the addition of the unsupported extensions (plpython3u, file_fdw)
- Remove objects that use the above items (perl_max, pymax)
- Remove other objects (**CAST with reference to pymax**)

**Note**: After each run of the script, delete the schema or replace all the items as appropriate when they already exist.

For some Azure Database for PostgreSQL service types, it is necessary to remove the following:

- All CREATE TEXT SEARCH items
- Rerun the query after removing all unsupported features.
- Note that the database dump may try to re-populate the public.mig_inventory table. We recommend that you remove the COPY ... FROM stdin statement. The mig_inventory table is used to inventory the objects that have been migrated to the target server.

**Note**: For help, reference the completed \artificats\azure_schema.sql file.

## Revert Server Parameters

With the migration completed, revert the server parameters of both instances to support the workload. Follow the steps in Server Parameters Reset.

## Summary - Backup and Restore

Even a simple backup and restore operation can potentially require significant effort to restore to an Azure Database for PostgreSQL instance.

Practice the above steps and document any items that were removed from the backup script during the actual migration. Record the time it takes to complete the entire migration.

# Data Migration with PostgreSQL COPY Command

## Setup

Follow all the steps in the setup guide to create an environment to support the following steps.

## Configure Server Parameters

To support the migration, set the source and target PostgreSQL instance parameters to allow for faster egress and ingress. Follow the steps in data-import-and-export.

## Install PgBouncer

To gain the performance benefits of PgBouncer, follow all the steps in the pgbouncer guide to install PgBouncer. Replace all references to port 5432 with port 6432.

## Schema

Typically, a COPY-based migration requires the target to contain the migrated schema. Follow the steps in the data-migration---schema document.

## Data

For larger databases with many tables and small migration windows, parallelize the import using a scripting tool. Example of tools are PowerShell and the COPY command.

### PowerShell and COPY command example script

- Switch to the **PREFIX-vm-pgdb01** virtual machine
- Create a new PowerShell script in c:\temp\migrate_copy.ps1 using the code below, update the PowerShell script with the local instance and the PostgreSQL target instance information.

```
function GetData($connString, $sql, $outputFile)
{
   $env:PGPASSWORD = $connString.Password;

   Write-Host "$global:psqlPath\psql.exe -h $($connString.Host) -p $($connString.Port) -U $($connString.User) -d
$($connstring.DbName) -c `"$sql;`"";

   if ($connString.useSSL)
   {
      $env:PGSSLMODE = "require";
   }
```

```
    $data = $sql | & $global:psqlPath\psql.exe -h $($connString.Host) -p $($connString.Port) -U $($connString.Us
er) -d $($connstring.DbName) -c `"$sql`";

    return $data;
}

function  ExportAllTables($source)
{
    #get the list of tables...
    $sql = "select table_name, table_schema from information_schema.tables where table_type = 'BASE TABLE' and
table_schema = '$($source.dbname)' order by table_name;";
    $data = GetData $source $sql $outputFile;

    foreach($item in $data)
    {
        if ($item.contains("table_name") -or $item.contains("rows)") -or $item.contains("-+-"))
        {
            continue;
        }

        $tableName, $schemaName = $item.split('|');

        if ( $tableName)
        {
            $tableName = $tableName.replace(" ","");
            $schemaName = $schemaName.replace(" ","");
            $fullName = $schemaName + "." + $tablename;

            mkdir "c:/export/$schemaName" -ea SilentlyContinue;

            $sql = "COPY (SELECT * FROM $fullName) TO 'c:/export/$schemaName/$($tableName).copy';";

            #export via psql...
            $data = GetData $source $sql $outputFile;
        }
    }
}

function ImportAllTables($source, $target)
{
    #get the list of tables...
    $sql = "select table_name, table_schema from information_schema.tables where table_type = 'BASE TABLE' and
```

```
table_schema = '$($source.dbname)' order by table_name;";
    $data = GetData $source $sql $outputFile;

    foreach($item in $data)
    {
        if ($item.contains("table_name") -or $item.contains("rows)") -or $item.contains("-+-"))
        {
            continue;
        }

        $tableName, $schemaName = $item.split('|');

        if ( $tableName)
        {
            $tableName = $tableName.replace(" ","");
            $schemaName = $schemaName.replace(" ","");
            $fullName = $schemaName + "." + $tablename;

            #truncate the table first...
            $sql = "truncate table $fullName;";
            $res = GetData $target $sql $outputFile;

            #import the data...
            $sql = "\COPY $fullName FROM 'c:/export/$schemaName/$($tableName).copy';";
            $res = GetData $target $sql $outputFile;
        }
    }
}

$global:psqlPath = "C:\Program Files\PostgreSQL\10\bin";

$sourceConnStringA = @{};
$sourceConnStringA.Host = "localhost";
$sourceConnStringA.Port = "5432";
$sourceConnStringA.User = "postgres";
$sourceConnStringA.Password = "Seattle123";
$sourceConnStringA.DbName = "reg_app";
$sourceConnStringA.UseSSL = $false;

$targetConnStringA = @{};
$targetConnStringA.Host = "servername.postgres.database.azure.com";
$targetConnStringA.Port = "5432";
$targetConnStringA.User = "s2admin@servername";
```

```
$targetConnStringA.Password = "Seattle123Seattle123";
$targetConnStringA.DbName = "reg_app";
$targetConnStringA.UseSSL = $true;
```

- Run the PowerShell script to start the migration

```
. c:\temp\migrate_copy.ps1

ExportAllTables $sourceConnStringA;

ImportAllTables $sourceConnStringA $targetConnStringA;
```

## Revert Server Parameters

With the migration completed, revert the server parameters of both instances to support the workload. Follow the steps in revert-server-parameters.

### Enable Keys and Triggers

As the final step, run the SQL to add all foreign keys and enable all triggers. Refer to the Schema Migration document for more information.

# Data Migration with Azure Data Factory (ADF)

## Setup

Follow all the steps in the Setup guide to create an environment to support the following steps.

**Note**: If your server is not enabled for SSL, ADF will not connect to server until it is configured. See configure-postgresql-ssl for SSL connectivity to enable SSL.

## Configure Server Parameters

To support the migration, set the source and target PostgreSQL instance parameters to allow for faster egress and ingress. Follow the steps in server-parameters.

## Install PgBouncer

To gain the performance benefits of PgBouncer, follow all the steps in the pgbouncer guide to setup PgBouncer. Replace all references to port 5432 with port 6432.

## Configure Network Connectivity

Even though the virtual network is allowing traffic on port 5432 (via the NSG deployed with the VM), the Windows Firewall may not be set to allow traffic. Run the following command on the Virtual Machine to open port 5432 and 6432.

```
netsh advfirewall firewall add rule name="TCP Port 5432" dir=in action=allow protocol=TCP localport=5432
netsh advfirewall firewall add rule name="TCP Port 6432" dir=in action=allow protocol=TCP localport=6432
```

## Schema

ADF will require the target Azure Database for PostgreSQL instance to contain the migrated schema. Follow the steps in the data-migration---schema document.

Verify foreign keys and triggers have been disabled.

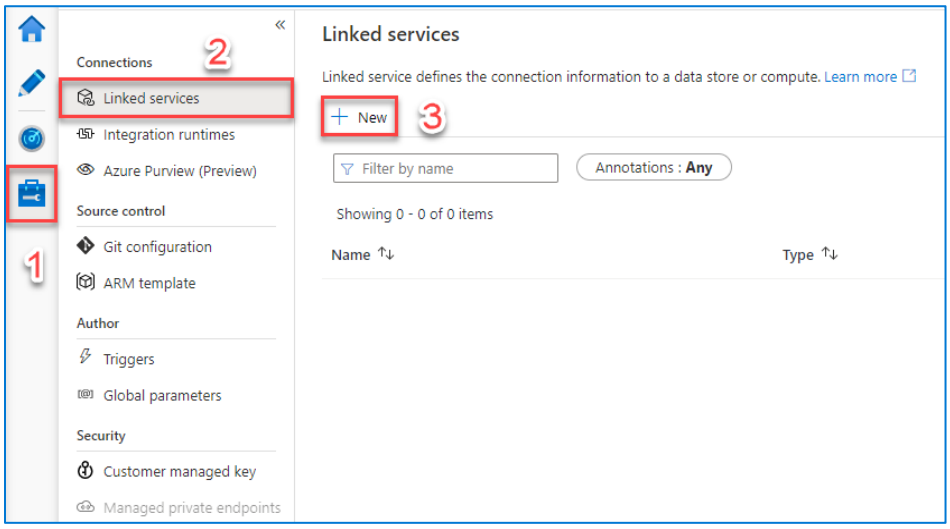
## Data

With the database objects and users from the source system migrated, begin the migration. ADF supports PostgreSQL versions from 7.4 and greater.

### Add ADF Linked Services

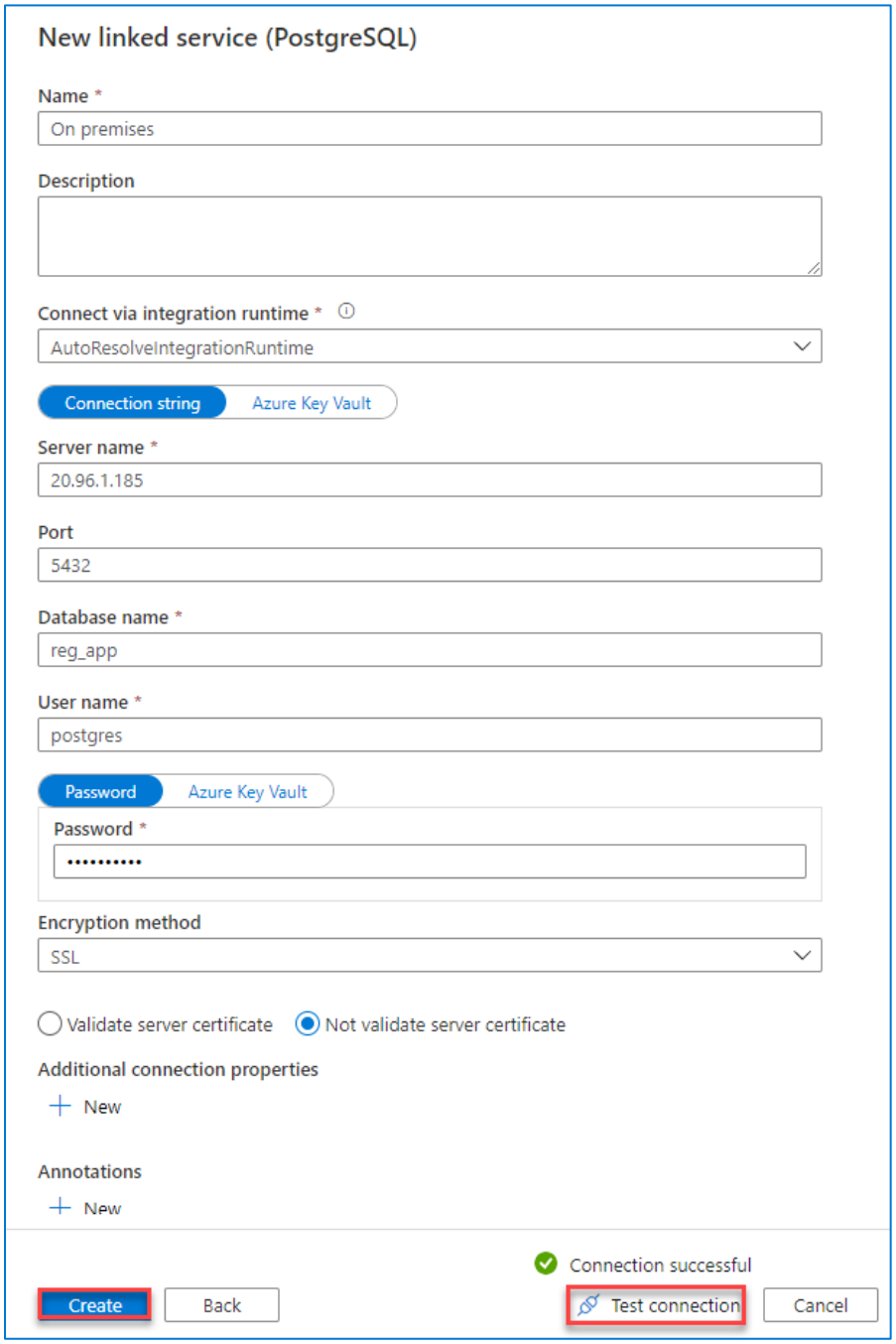
- Browse to the Azure Data Factory instance deployed from the ARM template

- On the **Overview** page, select **Author & Monitor**
- In the new ADF browser tab, in the left navigation, select the **Manage** tab
- For the **Linked services**, select **+ New**




Configuring a new linked service in the Data Factory Manage Hub.

- For the service type, select **PostgreSQL**, then select **Continue**
- For the name, type **OnPremises**
- For the server name, type the IP address of the Virtual Machine **PREFIX-vm-pgdb01**
- For the database name, type **reg_app**
- For the username, type **postgres**
- For the password, type **Seattle123**
- Set the encryption method to **SSL**
- Select to **Not validate server certificate**
- Feel free to test the connection before continuing
- Select **Create**

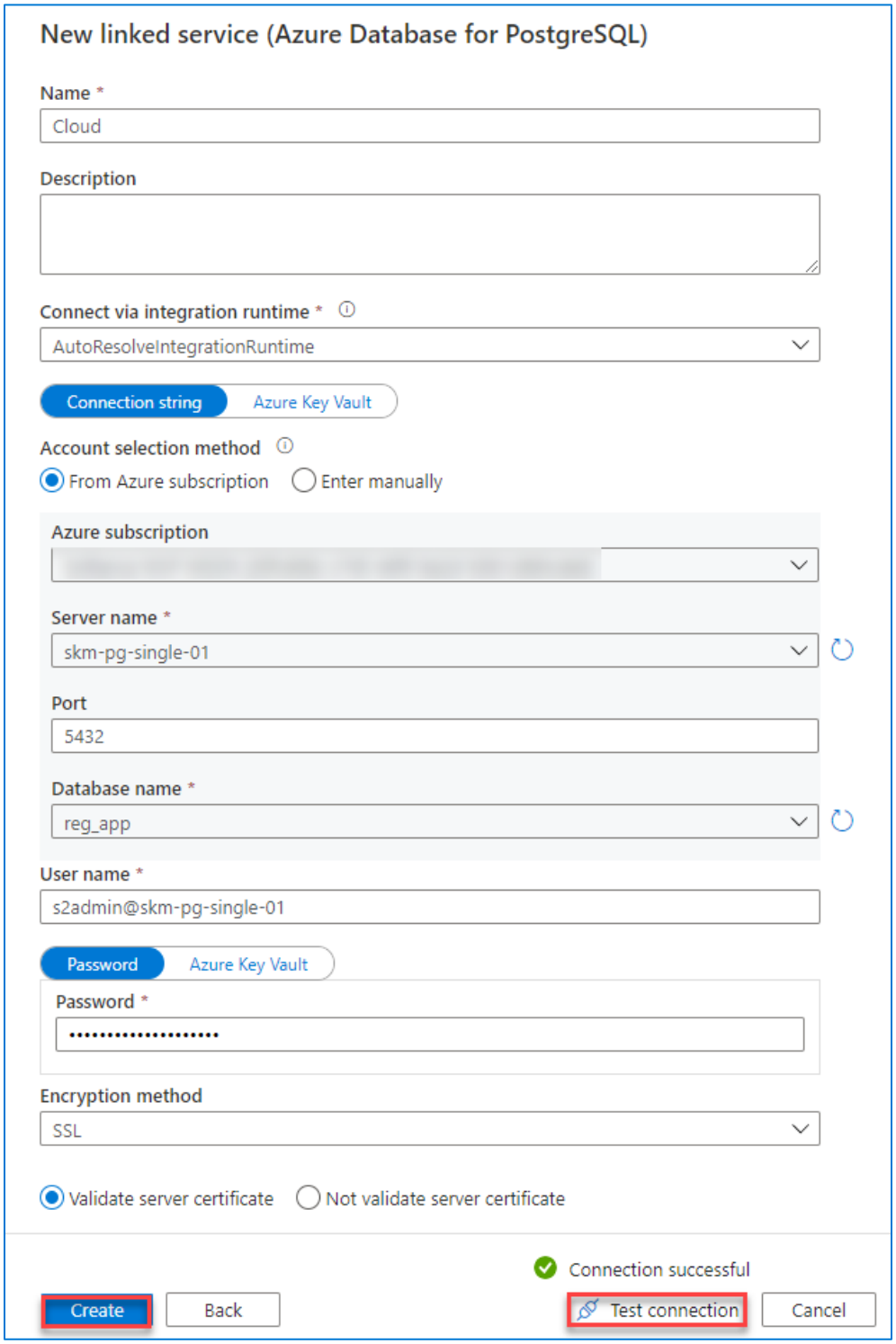



Referencing the on-premises PostgreSQL instance from Azure Data Factory and testing the connection.

**Note**: By default, **Azure VMs have dynamic public IP addresses**. If a VM is deallocated, the public IP will change the next time it is started. The issue can be bypassed by configuring a fully qualified domain name (FQDN) for your VM's public IP and then setting the **Server name** to that FQDN. Learn more about configuring a FQDN for your VM's public IP here.

For the **Linked services**, select **+ New**.

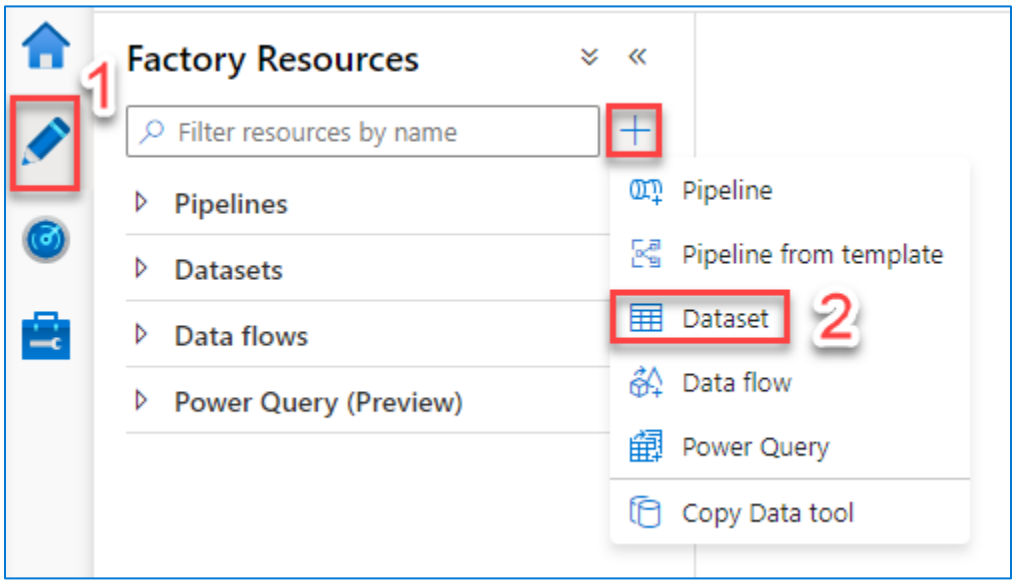
- For the service type, select **Azure Database for PostgreSQL**, then select **Continue**.
- For the name, type **Cloud**.
- For the server name, select the **PREFIX-pg-single-01** instance.
- For the database name, select **reg_app**.
- For the username, type **s2admin@PREFIX-pg-single-01**.
- For the password, type **Seattle123Seattle123**.
- For the encryption method, select **SSL**.
- Select **Validate server certificate**.
- Select **Test connection** and wait for the green **Connection successful** message.
- Select **Create**.




Referencing the Azure Database for PostgreSQL instance as a linked service and testing the connection.

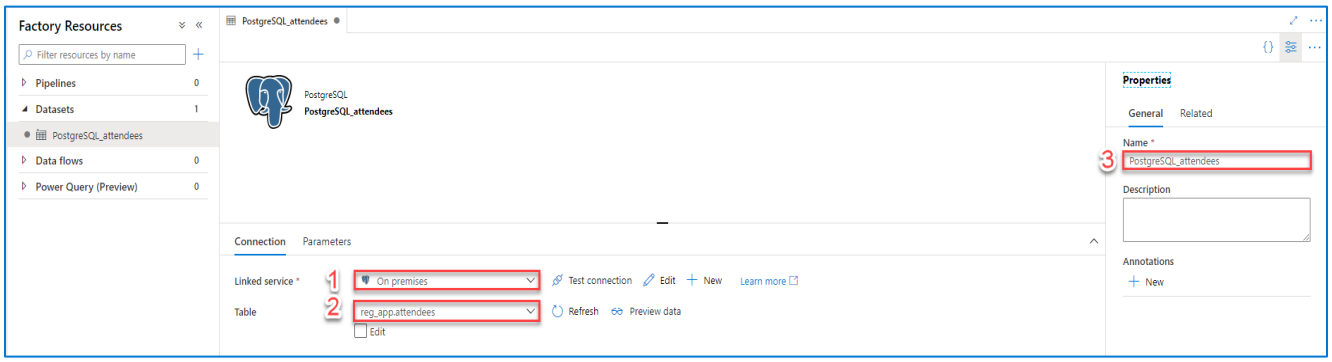
## Create Data Sets

- Select the **Author** tab.
- Select **+->Dataset**.




Creating a new dataset in the Author hub.

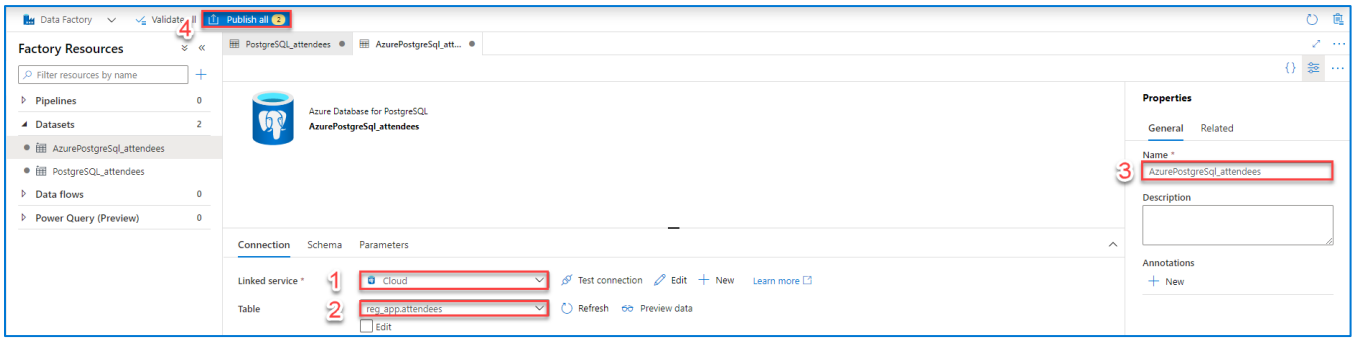
- Search for **PostgreSQL**, select it, and then select **Continue**.
- For the linked service, select the OnPremises PostgreSQL linked service.
- For the table, select **reg_app.attendees**.
- For the name, type **PostgreSQL_attendees**.




Selecting the attendees table in the on-premises PostgreSQL instance for the PostgreSQL_attendees dataset.

- Search for **Azure Database for PostgreSQL**, select it, and then select **Continue**.

- Select **+->Dataset**.
- For the type, select **Azure Database for PostgreSQL**.
- For the linked service, select the **Cloud** linked service.
- For the table, select **reg_app.attendees**.
- For the name, type **AzurePostgreSQL_attendees**.
- In the top navigation, select **Publish all**, then select **Publish**.

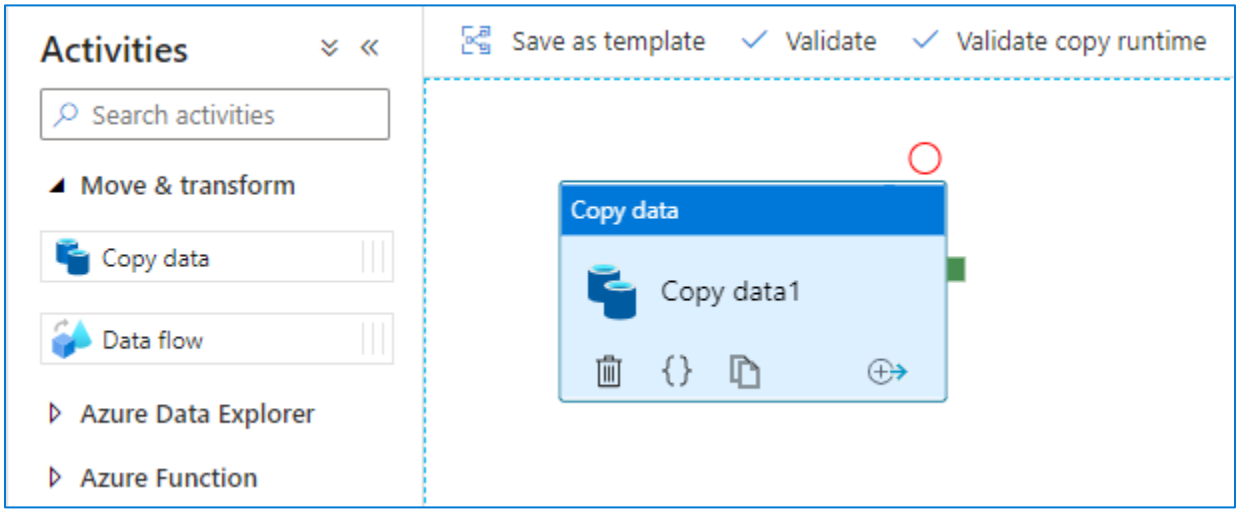



Selecting the attendees table in the Azure PostgreSQL instance for the AzurePostgreSQL_attendees dataset.

## Create Pipelines

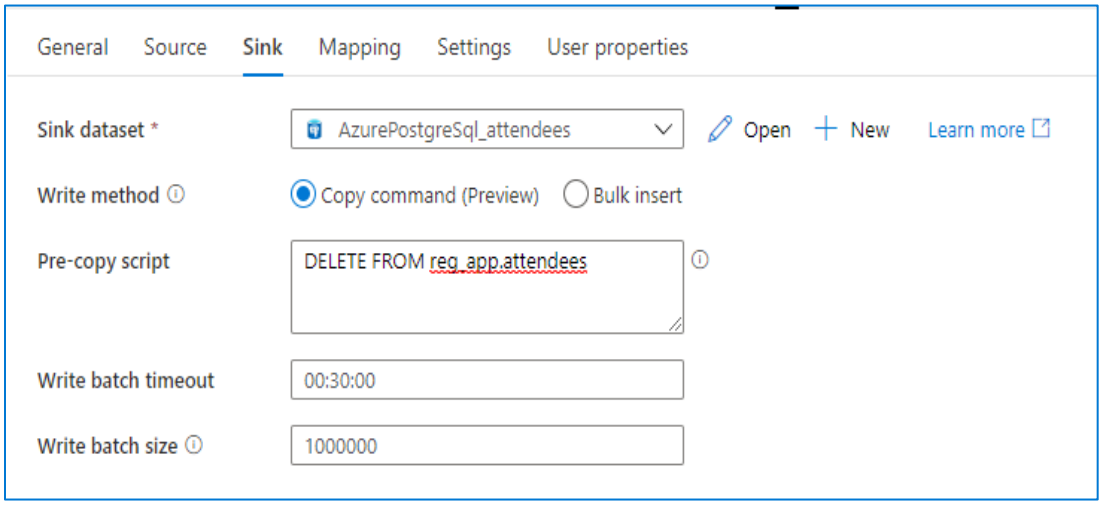
With the linked services and data sets in place, create a pipeline from the source to the destination.

- Select the **Author** tab.
- Select **+->Pipeline**.
- For the name, type **PostgreSQL_to_AzurePostgreSQL**.
- Drag a **Copy data** transformation to the work area (expand **Move & transform**).




Adding a Copy Data transformation to the PostgreSQL_to_AzurePostgreSQL pipeline.

- Select the **Source** tab.
- For the source dataset, select the **PostgreSql_attendees** data set. Leave other settings at their defaults.
- Select the **Sink** tab.
- For the sink dataset, select the **AzurePostgreSql_attendees** data set.
- For the **Pre-copy script**, type **delete from reg_app.attendees**.

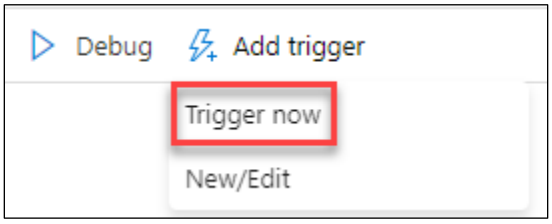



Configuring the sink settings for the Copy Data transformation with a pre-copy script.

- In the top navigation, select **Publish all** and **Publish**.

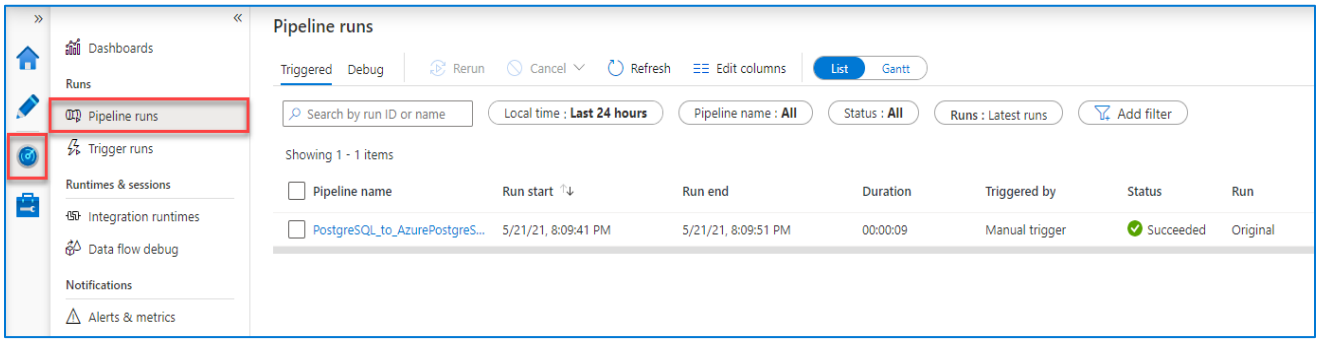
## Test the Pipeline

- Select **Add trigger->Trigger now**.




Adding a trigger to the new pipeline for testing.

- Select **OK**.
- Select the **Monitor** tab. Observe the displayed pipeline run:



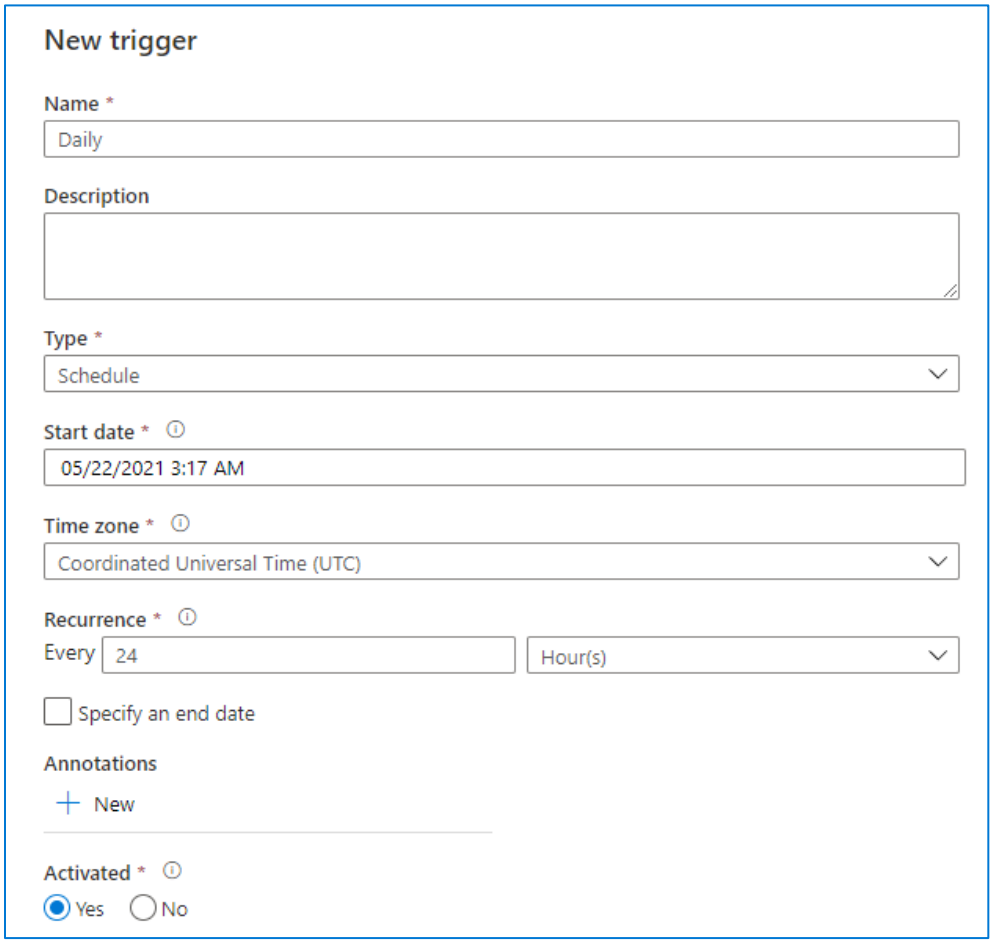

Observing the current pipeline run in the Monitor tab.

- If the pipeline run has an error, click the **error** icon to display the error. Resolve any errors.

## Create Triggers

For a continuous migration, create a trigger to execute it on a schedule.

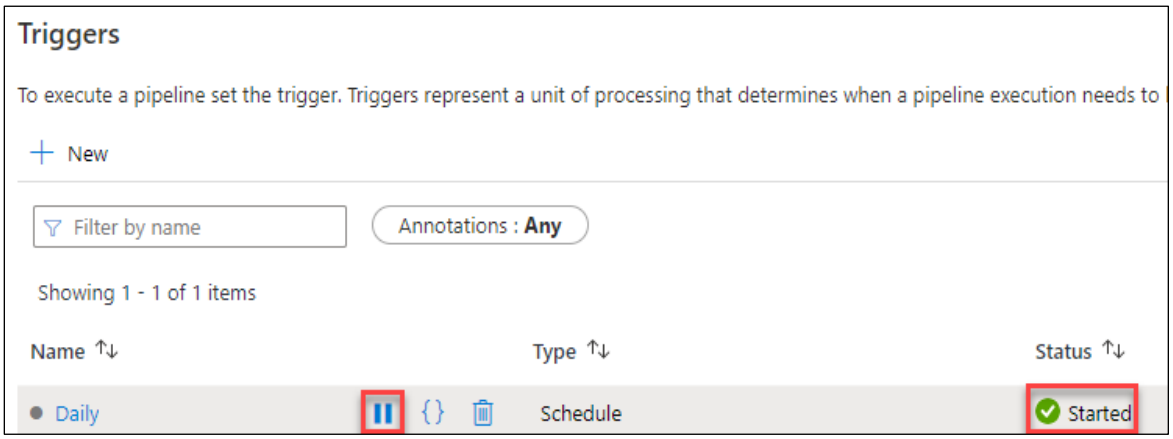
- Select the **Manage** tab.
- Select **Triggers**.
- Select **+ New**.
- For the name, type **Daily**.
- For the type, select **Schedule**.
- For the recurrence, select **24 Hours**.
- Select **OK**.




Create a trigger in the Manage hub that fires once daily.

- In the top navigation, select **Publish all** and **Publish**
- Select the **Author** tab
- Select the **PostgreSQL_to_AzurePostgreSQL** pipeline
- Select **Add trigger->New/Edit**
- In the dropdown, select the **Daily** trigger
- Select **OK**
- Select **OK**
- In the top navigation, select **Publish all** and **Publish**

**Note**: You may need to enable the trigger. Go to the **Manage** hub, enable the **Daily** trigger, and then publish the changes. The image below displays the trigger enabled message. However, *you should fully disable the trigger once you have migrated!*




Enabling the Daily trigger in the Manage hub.

### Optimal pipeline script example

The above series of steps implements a very basic and brute force migration path (delete all records and copy all records) with ADF. A more optimal version would perform change tracking and only sync the changed records from the source. However, this would require implementing extra steps, which would add more pipeline complexity.

For example, rather than a static table-based pipeline, create a lookup that queries for all tables and then loops through them to import the table data. To do this, create two new data sources and a pipeline that leverages them.

- Data sets (On-premises)

```
{
  "name": "PostgreSql_generic",
  "properties": {
    "linkedServiceName": {
      "referenceName": "PostgreSql1",
      "type": "LinkedServiceReference"
    },
    "parameters": {
      "DWTableName": {
        "type": "string"
      },
      "DWSchema": {
        "type": "string"
      }
```

```
        },
        "annotations": [],
        "type": "PostgreSqlTable",
        "schema": [],
        "typeProperties": {
            "schema": {
                "value": "@dataset().DWSchema",
                "type": "Expression"
            },
            "table": {
                "value": "@dataset().DWTableName",
                "type": "Expression"
            }
        }
    },
    "type": "Microsoft.DataFactory/factories/datasets"
}
```

- Data sets (Azure)

```
{
    "name": "AzurePostgreSql_generic",
    "properties": {
        "linkedServiceName": {
            "referenceName": "AzurePostgreSql1",
            "type": "LinkedServiceReference"
        },
        "parameters": {
            "DWTableName": {
                "type": "string"
            },
            "DWSchema": {
                "type": "string"
            }
        },
        "annotations": [],
        "type": "AzurePostgreSqlTable",
        "schema": [],
        "typeProperties": {
            "schema": {
                "value": "@dataset().DWSchema",
                "type": "Expression"
            },
```

```
            "table": {
                "value": "@dataset().DWTableName",
                "type": "Expression"
            }
        }
    },
    "type": "Microsoft.DataFactory/factories/datasets"
}
```

- Pipeline

```
{
    "name": "PostgreSQL_to_AzurePostgreSQL",
    "properties": {
        "activities": [
            {
                "name": "Lookup1",
                "type": "Lookup",
                "dependsOn": [],
                "policy": {
                    "timeout": "7.00:00:00",
                    "retry": 0,
                    "retryIntervalInSeconds": 30,
                    "secureOutput": false,
                    "secureInput": false
                },
                "userProperties": [],
                "typeProperties": {
                    "source": {
                        "type": "PostgreSqlSource",
                        "query": "  select table_name, table_schema\n  from information_schema.tables\n  where table_sche
ma = 'reg_app' "
                    },
                    "dataset": {
                        "referenceName": "PostgreSql_attendees",
                        "type": "DatasetReference",
                        "parameters": {
                            "DWTableName": "tables",
                            "DWSchema": "information_schema"
                        }
                    },
                    "firstRowOnly": false
                }
```

```
        },
        {
            "name": "ForEach1",
            "type": "ForEach",
            "dependsOn": [
                {
                    "activity": "Lookup1",
                    "dependencyConditions": [
                        "Succeeded"
                    ]
                }
            ],
            "userProperties": [],
            "typeProperties": {
                "items": {
                    "value": "@array(activity('Lookup1').output.value)",
                    "type": "Expression"
                },
                "isSequential": true,
                "activities": [
                    {
                        "name": "Copy data1_copy1",
                        "type": "Copy",
                        "dependsOn": [],
                        "policy": {
                            "timeout": "7.00:00:00",
                            "retry": 0,
                            "retryIntervalInSeconds": 30,
                            "secureOutput": false,
                            "secureInput": false
                        },
                        "userProperties": [],
                        "typeProperties": {
                            "source": {
                                "type": "PostgreSqlSource"
                            },
                            "sink": {
                                "type": "AzurePostgreSQLSink",
                                "writeBatchSize": 1000000,
                                "writeBatchTimeout": "00:30:00",
                                "writeMethod": "BulkInsert"
                            },
                            "enableStaging": false,
```

```
                "enableSkipIncompatibleRow": true,
                "translator": {
                    "type": "TabularTranslator",
                    "mappings": [
                        {
                            "source": {
                                "name": "id",
                                "type": "Int32"
                            },
                            "sink": {
                                "name": "id",
                                "type": "Int32",
                                "physicalType": "integer"
                            }
                        },
                        {
                            "source": {
                                "name": "first_name",
                                "type": "String"
                            },
                            "sink": {
                                "name": "first_name",
                                "type": "String",
                                "physicalType": "character varying"
                            }
                        },
                        {
                            "source": {
                                "name": "last_name",
                                "type": "String"
                            },
                            "sink": {
                                "name": "last_name",
                                "type": "String",
                                "physicalType": "character varying"
                            }
                        },
                        {
                            "source": {
                                "name": "email_address",
                                "type": "String"
                            },
                            "sink": {
```

```
                        "name": "email_address",
                        "type": "String",
                        "physicalType": "character varying"
                    }
                }
            ],
            "typeConversion": true,
            "typeConversionSettings": {
                "allowDataTruncation": true,
                "treatBooleanAsNumber": false
            }
        }
    },
    "inputs": [
        {
            "referenceName": "PostgreSql_generic",
            "type": "DatasetReference",
            "parameters": {
                "DWTableName": "@item().table_name",
                "DWSchema": "@item().table_schema"
            }
        }
    ],
    "outputs": [
        {
            "referenceName": "AzurePostgreSql_generic",
            "type": "DatasetReference",
            "parameters": {
                "DWTableName": {
                    "value": "@item().table_name",
                    "type": "Expression"
                },
                "DWSchema": "@item().table_schema"
            }
        }
    ]
}
]
}
}
],
"annotations": []
```

```
    }
}
```

- To create these resources, you will need to use PowerShell. For example, save the first dataset definition in c:\temp\onprem-dataset.json. From the c:\temp directory, execute the following:

```
$dataFactoryName = "[DATA FACTORY NAME]"
$resourceGroup = "[RESOURCE GROUP]"
$datasetName = "PostgreSql_generic"

Set-AzDataFactoryV2Dataset -DataFactoryName $dataFactoryName `
    -ResourceGroupName $resourceGroup -Name $datasetName `
    -DefinitionFile ".\onprem-dataset.json"
```

- The pipeline follows a similar command:

```
Set-AzDataFactoryV2Pipeline `
    -DataFactoryName $dataFactoryName `
    -ResourceGroupName $resourceGroup `
    -Name $datasetName `
    -DefinitionFile ".\migration-pipeline.json"
```

- Consult this document for a more comprehensive PowerShell example.

## Revert Server Parameters

With the migration completed, revert the server parameters to support the workload. Follow the steps in revert-server-parameters.

### Enable Keys and Triggers

As the final step, run the SQL to reset all foreign keys and to enable all triggers.

# Data Migration with Azure Database Migration Service (DMS)

## Setup

Follow all the steps in the setup guide to create an environment to support the following steps.

**Note**: Enable SSL on the server. The DMS will not be usable until you configure it. See configure-postgresql-ssl to enable SSL for your instance.

## Configure Server Parameters

To support the migration, set the source and target PostgreSQL instance parameters to allow for faster egress and ingress. Follow the steps in server-parameters.

**Note**: Specifically, set the wal_level parameter to replica for DMS.

## Install PgBouncer

To gain the performance benefits of PgBouncer, follow all the steps in the pgbouncer guide to setup PgBouncer. Replace all references to port 5432 with port 6432.

## Configure Network Connectivity

Even though the virtual network is allowing traffic on port 5432, the Windows Firewall may not be configured to allow traffic. Run the following command on the Virtual Machine to open port 5432 and 6432:

```
netsh advfirewall firewall add rule name="TCP Port 5432" dir=in action=allow protocol=TCP localport=5432
netsh advfirewall firewall add rule name="TCP Port 6432" dir=in action=allow protocol=TCP localport=6432
```

## Schema

ADF will require the target Azure Database for PostgreSQL instance to contain the migrated schema. Follow the steps in the data-migration---schema document.

## Data

### Tool Choice

With the database objects and users from the source system migrated, begin the migration.

**Note**: Databases running on PostgreSQL versions lower than 10.0 cannot use Azure DMS to migrate the workload. Instead, migrate users using the PostgreSQL pgAdmin or other tools.

### Database Migration Service

- In the Azure Portal, navigate to the Azure Database Migration Service.
- If the service is **Stopped**, click **Start Service** in the top navigation menu. Wait for the service to start.

**Note**: This can take a few minutes.

- In the top navigation, select **New Migration Project**.
- For the **name**, type **postgres_to_azure.**
- For the **source server type**, select **PostgreSQL**.
- For the migration type, select **Online data migration**.

**Note**: Create a DMS instance in the **premium** tier to support online migrations.

# New migration project ...

A database migration project is a group of database activities that you can migrate together.

**Migration project name**

Project name * postgres_to_azure

**Choose your source and target server type.**

Source server type * PostgreSQL

Target server type * Azure Database for PostgreSQL

**Choose your migration activity type.**

Migration activity type * Online data migration

Use this option to migrate databases that must be accessible and continuously updated during migration.

To continuously replicate data changes from your source to your target, please implement the steps below on your source server. These steps can be implemented anytime between the moment you create a project and the moment you start a migration activity. After your migration is complete, you will reverse those preparation steps.

- The source SQL Server version must be SQL Server 2005 or above.
- Database(s) must be in either Bulk-logged or Full recovery mode.
- Make sure to take full database backups.
- If tables do not have a primary key, enable CDC on the database and specific tables.
- The distributor role must be configured for the source database.

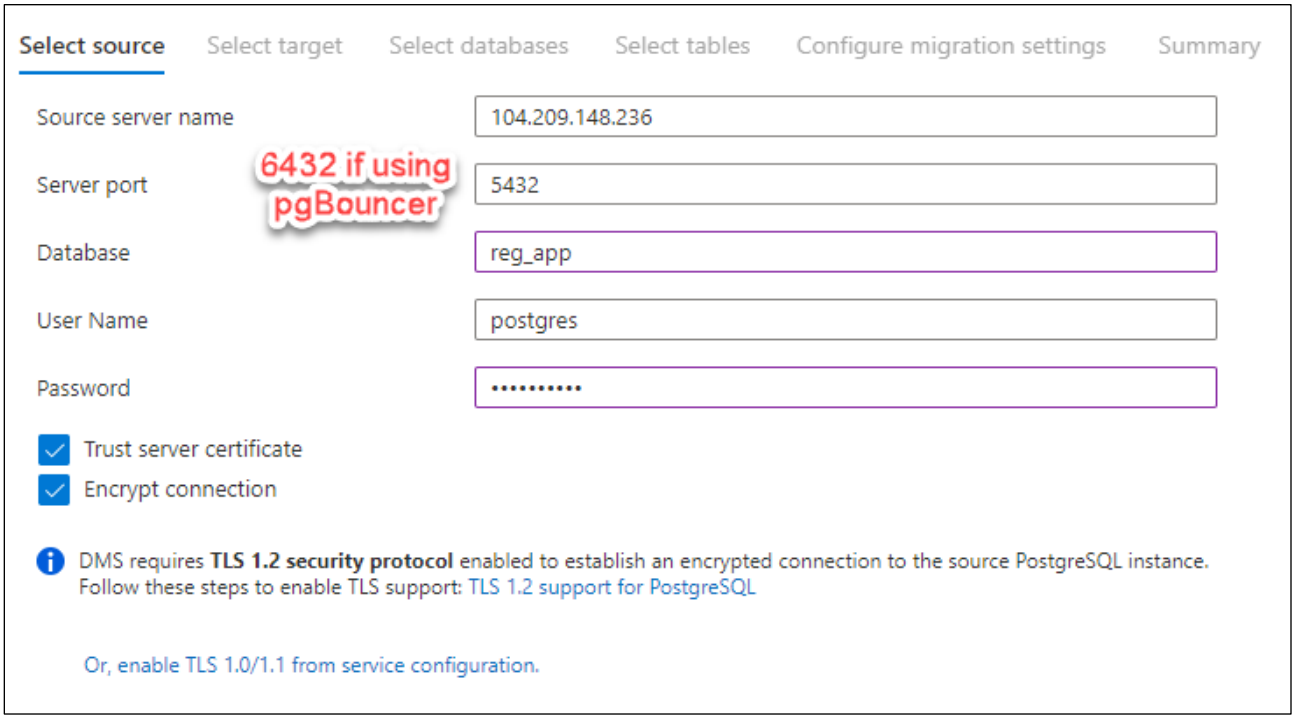
**To successfully use Database Migration Service (DMS) to migrate data, you need to:**

1. Migrate schema using pg_dump -o -h hostname -U db_username -d db_name -s > your_schema.sql
2. Remove foreign keys in schema at target Azure Database for PostgreSQL
3. Disable triggers at target Azure Database for PostgreSQL

Create and run activity

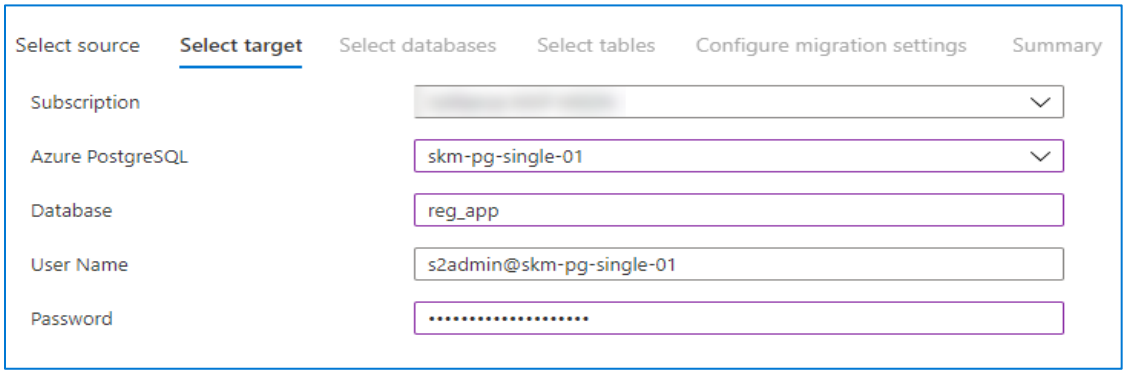
Creating a new online migration in DMS.

- Select **Create and run activity**.
- For the source server, type the **PREFIX-vm-pgdb01** Virtual Machine public IP address (you can also use the VM public IP FQDN, if you enabled it).
- For the database, type **reg_app.**
- For the username, type **postgres.**
- For the password, type **Seattle123.**
- Trust the server certificate.



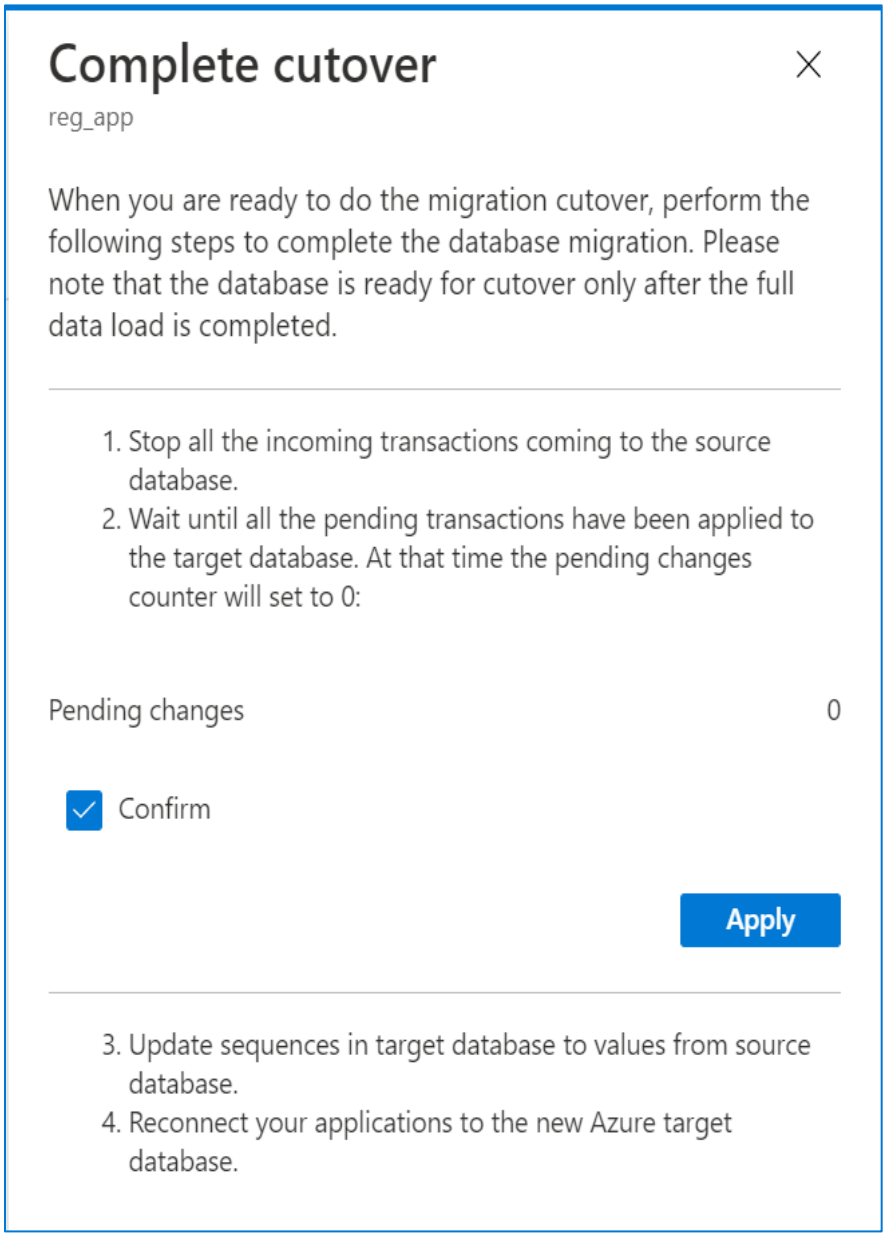

Configuring the on-premises PostgreSQL instance as the migration source.

- Select **Next: Select target>>**.
- Select the target subscription.
- Select the Azure Database for PostgreSQL instance.
- For the username, type **s2admin@servername.**
- For the password, type **Seattle123Seattle123.**




Configuring the Azure PostgreSQL database as the migration landing zone.

- Select **Next: Select databases>>**.
- Select the **reg_app** database.
- Select **Next: Select Tables**.
- Select all the tables, except the **reg_app.ddl_history** and **public.mig_inventory** tables.
- Select **Next: Configure migration settings>>**.
- Set the maximum number of tables to load in parallel to **50**.
- Select **Next: Summary>>**.
- For the name, type **MigratePostgreSQL.**
- Select **Start migration**, the page will update with the migration status.
- Once the migration status changes to **Ready to cutover**, select the target database and select **Start cutover**. You will need to wait for all incoming transactions to complete before the cutover can commence.
- Select **Confirm**, then select **Apply**.




Performing the migration cutover once transactions have reached 0.

## Revert Server Parameters

With the migration completed, revert the server parameters to support the workload. Follow the steps in revert-server-parameters.

### Enable Keys and Triggers

As the final step, run the SQL to reset all foreign keys and to enable all triggers.

When following the steps in the Schema Migration document, there should exist a collection of scripts in the C:\temp folder. Execute the following commands in the command prompt.

**Note**: The first command will enable triggers, while the second command will add constraints (foreign keys).

```
psql --host=[PREFIX]-pg-single-01.postgres.database.azure.com --port=5432 --username=postgres@[PREFIX]-pg-single-01 --dbname=reg_app --file=c:\temp\enable_triggers.sql

psql --host=[PREFIX]-pg-single-01.postgres.database.azure.com --port=5432 --username=postgres@[PREFIX]-pg-single-01 --dbname=reg_app --file=c:\temp\add-constraints.sql
```

# Data Migration with PostgreSQL Logical Replication

## Setup

Follow all the steps in the setup guide to create an environment to support the following steps.

**Note**: If the server is not enabled for SSL, it is highly recommended that it is configured. See Configure PostgreSQL for SSL connectivity to enable SSL for the instance.

## Replication setup

Logical replication will send only the changes that occur in real-time to the subscriber. This means an initial synchronization will be required before enabling the replication.

### Backup and Restore the database

- Using the steps in the backup-and-restore, backup the database and restore it to the flex server.

**Note**: Remove any unsupported features before the restore.

### Set the wal_level

- On the source PostgreSQL instance, edit the postgresql.conf file.
- Set the wal_level server parameter to **logical** and save the file.
- From a command line, restart the source instance.

```
net stop postgresql-x64-10
net start postgresql-x64-10
```

### Create a replication User

- Open PostgreSQL pgAdmin and connect as the local database's superuser.
- Run the following command:

```
create user replicationuser with REPLICATION encrypted password 'Seattle123';
```

### Grant permissions to the replication user

- Grant the replication user access to the tables that will be replicated. Run the following commands:

```
GRANT ALL PRIVILEGES ON DATABASE reg_app TO replicationuser;

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO replicationuser;
```

### Create the Publication

- In the pgAdmin tool on the source instance, run the following commands:

```
CREATE PUBLICATION PUB_REPLICATION;
```

- Add some tables to the publication

```
ALTER PUBLICATION PUB_REPLICATION ADD TABLE reg_app.attendees;
ALTER PUBLICATION PUB_REPLICATION ADD TABLE reg_app.registrations;
```

### Create Flexible Server Connection

- In PostgreSQL pgAdmin, create a new connection to the Azure Database for PostgreSQL Flexible Server.
  - For Name, enter a name such as **Azure PostgreSQL Flex**
  - Select the **Connection** tab.
  - For Hostname, enter the full server DNS (ex: PREFIX-pg-flex-01.postgres.database.azure.com).
  - Enter the username (ex: s2admin with no servername).
  - Enter the password (**Seattle123Seattle123**).
  - Select the **SSL** tab.
  - For the Root certificate, browse to the **BaltimoreCyberTrustRoot.crt.pem** key file.
  - Select **Save**.

### Create the Subscription

- Switch to the pgAdmin tool and select the query window with the Azure Database for PostgreSQL Flexible Server connection.
- Run the following commands:

```
CREATE SUBSCRIPTION sub
   CONNECTION 'host={vmIP} port=5432 user=replicationuser dbname=reg_app connect_timeout=10 password=
Seattle123 sslmode=require'
   PUBLICATION PUB_REPLICATION
```

**Note**: Only Flexible Server supports this command. It will return a permissions error for any other offering.

**Note**: All tables in the publication must already exist in the target instance.

### Test the Publication and Subscription

- Execute the following query on the source instance:

```
Insert into REG_APP.ATTENDEES (ID,FIRST_NAME,LAST_NAME,EMAIL_ADDRESS) values (200,'Chris','Givens','chrisg@adventure-works.com');
```

- Run the following query on the target flexible server:

```
select * from REG_APP.ATTENDEES where last_name = 'Givens'
```

- Observe the change replicated to the flexible server. From now on, all INSERT, UPDATE, and DELETE queries will be replicated across servers unidirectionally.

**Note**: Write-based queries on replica servers are not replicated back to the master server. PostgreSQL currently has limited support for resolving conflicts when the data between servers diverges. If there is a conflict, the replication will stop and PostgreSQL will wait until the issue is manually fixed by the database administrator. Most applications will direct all write operations to the master server and distribute reads among available replica servers.

### Notes

As noted above, logical replication will only handle events such as INSERT, UPDATE, and DELETE. Any supporting artifacts, such as SEQUENCEs, require synchronization through some other process.

# Data Migration with PostgreSQL Logical Decoding

## Setup

Follow all the steps in the setup guide to create an environment to support the following steps.

- **Note**: Logical Decoding requires the target database to contain the migrated schema. Please refer to the data-migration---schemat the postgresql.conf (C:\Program Files\PostgreSQL\10\data\postgresql.conf) file.
- Set the wal_level server parameter to **logical** and save the file.
- From a command line, restart the source instance:

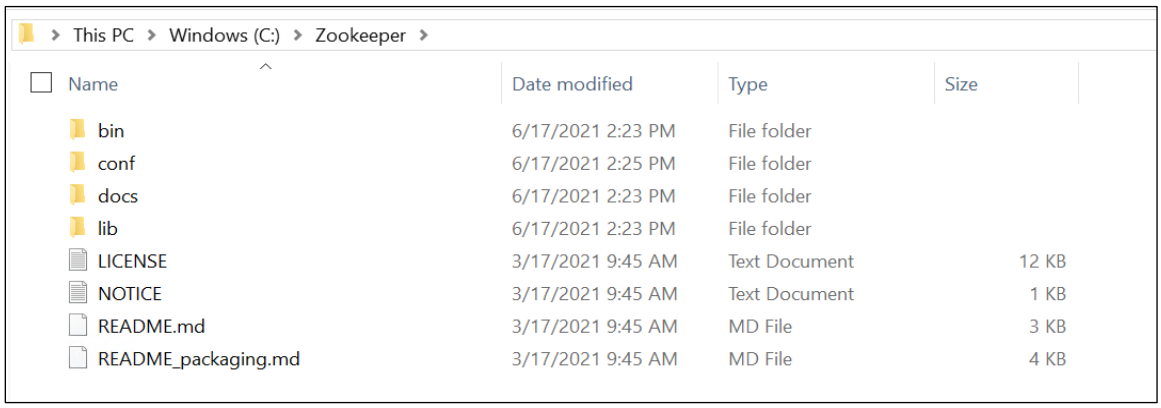
```
net stop postgresql-x64-10
net start postgresql-x64-10
```

### Extra Setup

As mentioned previously, logical decoding requires custom tools to monitor replication slots. In this example, we demonstrate how to configure Debezium and its dependencies (Kafka and Zookeeper) to get logical decoding working between a source and a target instance. Debezium extracts data from the source and will use a JDBC connector to send data to a sink.

#### Install Debezium Dependencies

- Download and Install 7zip
- Install Zookeeper
  - Download Zookeeper
  - Using 7zip, extract the gzip.
  - Using 7zip, extract the extracted tar file contents to c:\ZooKeeper. Ensure the bin, conf and other directories show in c:\zookeeper.

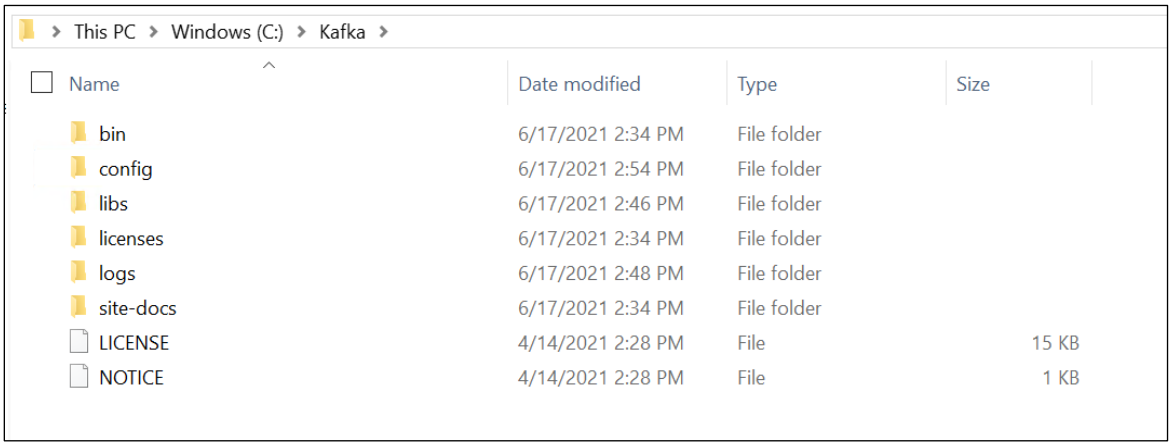


The contents of the Zookeeper folder.

- Copy **C:\zookeeper\conf\zoo_sample.cfg** and rename it to **zoo.cfg**
- Edit the zoo.cfg file.
- Update the dataDir to point to c:\zookeeper\data.
- Add the ZOOKEEPER_HOME system environment variable. Set it to c:\zookeeper.
- Add c:\zookeeper\bin to the PATH environment variable.

  **Note**: Be mindful to edit System environment variables, rather than User ones. For a review on editing environment variables, please consult the setup.

- Install Kafka
  - Download Kafka
  - Using 7zip, extract the zipfile to c:\Kafka.




The contents of the Kafka folder, extracted from the archive.

  - Edit the c:\kafka\config\zookeeper.properties file.
  - Update the dataDir to point to c:\kafka\zookeeper-data.
  - Add c:\kafka\bin to the PATH environment variable.

- Install Debezium PostgreSQL connector
  - Download the Postgres Connector
  - Extract the folder to c:/temp/postgre-connector.

  - Copy the jar files to the c:/kafka/libs directory.

- Install the Confluent JDBC sink connector
  - Download the connector
  - Using 7zip, extract the lib folder to the c:/kafka/libs directory.

    **Note**: You may be notified that a file called slf4j-api-1.7.30.jar already exists in the c:/kafka/libs folder. Feel free to replace the file in the destination.

## wal2json

You will need to download a pre-compiled version of wal2json or download the source and compile it.

**wal2json** will likely need a Visual Studio runtime installed depending on how it is compiled. You can use the Dependency Walker mentioned in the planning module to determine which version it is looking for and any other needed dll/modules that may be missing from your system. For the pre-compiled example above, it requires the vcruntime140d.dll and ucrtbased.dll modules.

## Replication setup

### Start services

- Open a command prompt and run the following to start ZooKeeper:

```
c:\kafka\bin\windows\zookeeper-server-start.bat c:\kafka\config\zookeeper.properties
```

- Open a new command prompt and run the following to start Kafka:

```
c:\kafka\bin\windows\kafka-server-start.bat c:\kafka\config\server.properties
```

**Note**: If you have to stop and restart the ZooKeeper and Kafka processes, you may also need to delete the C:\tmp\kafka-logs directory if you get cluster ID issues.

- Create a new file called c:\kafka\config\postgres.properties. You can use an editor, such as Visual Studio Code.

- Copy the following into it:

```
name=dbz-test-connector
connector.class=io.debezium.connector.postgresql.PostgresConnector
tasks.max=1
plugin.name=wal2json
database.hostname=localhost
database.port=5432
database.user=postgres
```

```
database.password=Seattle123
database.dbname =reg_app
database.server.name=localhost
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=false
value.converter.schemas.enable=false
```

- Open a new command prompt and run the following to start the Kafka Connector:

```
c:\kafka\bin\windows\connect-standalone.bat c:\kafka\config\connect-standalone.properties c:\kafka\config\postgres.properties
```

### Create and deploy the Connector configuration

- Create a new file c:\kafka\config\postgres.json.
- Add the following to it:

```
{
  "name": "postgresql-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "localhost",
    "plugin.name": "wal2json",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "Seattle123",
    "database.dbname" : "reg_app",
    "database.server.name": "localhost"
  }
}
```

- Run the following PowerShell command to add the connector config to Kakfa:

```
Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method POST -infile c:/kafka/config/postgres.json -Uri http://localhost:8083/connectors/ -ContentType application/json
```

**Note**: If you are unable to connect to the remote server, ensure that you started the Kafka connector.

- Here is the output of the previous command.

```
PS C:\Users\s2admin> Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method POST -infile c:/kafka/config/postgres.json
 -Uri http://localhost:8083/connectors/ -ContentType application/json


StatusCode        : 201
StatusDescription : Created
Content           : {"name":"postgresql-connector","config":{"connector.class":"io.debezium.connector.postgresql.Postgr
                    esConnector","database.hostname":"localhost","plugin.name":"wal2json","database.port":"5432","datab
                    as...
RawContent        : HTTP/1.1 201 Created
                    Content-Length: 418
                    Content-Type: application/json
                    Date: Thu, 17 Jun 2021 15:06:16 GMT
                    Location: http://localhost:8083/connectors/postgresql-connector
                    Server: Jetty(9.4.39.v2...
Forms             : {}
Headers           : {[Content-Length, 418], [Content-Type, application/json], [Date, Thu, 17 Jun 2021 15:06:16 GMT],
                    [Location, http://localhost:8083/connectors/postgresql-connector]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 418
```

This image demonstrates a PowerShell window with the results of calling the REST endpoint to add the PostgreSQL connector to Apache Kafka.

- To see if the connector was added, run the following:

```
Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method GET -Uri http://localhost:8083/connect
ors/
```

- Here is the expected output.

```
PS C:\Users\s2admin> Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method GET -Uri http://localhost:8083/connectors/


StatusCode        : 200
StatusDescription : OK
Content           : ["dbz-test-connector","postgresql-connector"]
RawContent        : HTTP/1.1 200 OK
                    Content-Length: 45
                    Content-Type: application/json
                    Date: Thu, 17 Jun 2021 15:12:18 GMT
                    Server: Jetty(9.4.39.v20210325)

                    ["dbz-test-connector","postgresql-connector"]
Forms             : {}
Headers           : {[Content-Length, 45], [Content-Type, application/json], [Date, Thu, 17 Jun 2021 15:12:18 GMT], [Server,
                    Jetty(9.4.39.v20210325)]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 45
```

Testing the REST endpoint in PowerShell to confirm that the PostgreSQL connector has been added.

### Create the JDBC sink connector

- Create a new file c:\kafka\config\postgresink.json.

- Add the following to it. Replace the servername placeholder:

```
{
  "name": "jdbc-sink",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
```

```
  "tasks.max": "1",
  "topics": "localhost.reg_app.attendees",
  "dialect.name": "PostgreSqlDatabaseDialect",
  "connection.url": "jdbc:postgresql://servername.postgres.database.azure.com:5432/reg_app?user=s2admin&pas
sword=Seattle123Seattle123",
  "transforms": "unwrap",
  "transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
  "transforms.unwrap.drop.tombstones": "false",
  "auto.create": "true",
  "insert.mode": "upsert",
  "pk.fields": "id",
  "pk.mode": "record_key",
  "delete.enabled": "true"
 }
}
```

- Run the following PowerShell command to add the connector config to Kakfa:

```
Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method POST -infile c:/kafka/config/postgresink
.json -Uri http://localhost:8083/connectors/ -ContentType application/json
```

- Observe the following output:

```
PS C:\Users\s2admin> Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method POST -infile c:/kafka/config/postgressink.json
 -Uri http://localhost:8083/connectors/ -ContentType application/json


StatusCode        : 201
StatusDescription : Created
Content           : {"name":"jdbc-sink","config":{"connector.class":"io.confluent.connect.jdbc.JdbcSinkConnector","tasks.max":"1",
                    "topics":"localhost.reg_app.attendees","dialect.name":"PostgreSqlDatabaseDialect","connect...
RawContent        : HTTP/1.1 201 Created
                    Content-Length: 653
                    Content-Type: application/json
                    Date: Thu, 17 Jun 2021 16:46:58 GMT
                    Location: http://localhost:8083/connectors/jdbc-sink
                    Server: Jetty(9.4.39.v20210325)
                    ...
Forms             : {}
Headers           : {[Content-Length, 653], [Content-Type, application/json], [Date, Thu, 17 Jun 2021 16:46:58 GMT], [Location,
                    http://localhost:8083/connectors/jdbc-sink]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 653
```

Calling the REST endpoint in PowerShell to specify the target database (sink).

## Create the file sink connector

- Create a new file c:\kafka\config\filesink.json.

- Add the following to it:

```
{
  "name": "cdc-file-sink",
  "config": {
```

```
    "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": "1",
    "topics": "localhost.reg_app.attendees",
    "file": "c:/temp/postgres.file.txt"
  }
}
```

- Run the following PowerShell command to add the connector config to Kafka:

```
Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method POST -infile c:/kafka/config/filesink.json
-Uri http://localhost:8083/connectors/ -ContentType application/json
```

```
PS C:\Users\s2admin> Invoke-WebRequest -Headers @{"Accept" = "application/json"} -Method POST -infile c:/kafka/config/filesink.json
-Uri http://localhost:8083/connectors/ -ContentType application/json


StatusCode        : 201
StatusDescription : Created
Content           : {"name":"cdc-file-sink","config":{"connector.class":"org.apache.kafka.connect.file.FileStreamSinkConnector","tas
ks
                    .max":"1","topics":"localhost.reg_app.attendees","file":"c:/temp/postgres.file.txt","n...
RawContent        : HTTP/1.1 201 Created
                    Content-Length: 285
                    Content-Type: application/json
                    Date: Thu, 17 Jun 2021 16:51:58 GMT
                    Location: http://localhost:8083/connectors/cdc-file-sink
                    Server: Jetty(9.4.39.v20210325...
Forms             : {}
Headers           : {[Content-Length, 285], [Content-Type, application/json], [Date, Thu, 17 Jun 2021 16:51:58 GMT], [Location,
                    http://localhost:8083/connectors/cdc-file-sink]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 285
```

Calling the REST endpoint in PowerShell to configure Logical Decoding file sink.

### Test the replication

- Run the following SQL scripts to make various changes to the source system:

```
Insert into REG_APP.ATTENDEES (ID,FIRST_NAME,LAST_NAME,EMAIL_ADDRESS) values (210,'Bill','Gates','billg@a
dventure-works.com');
```

- Run the following queries on the target to see that the changes were migrated:

```
select * from reg_app.attendees where last_name = 'Gates'
```

- Review the sink file c:\temp\postgres.file.txt for all the queued changes.

### Helpful References

- Documentation on the Debezium connector
- To implement a custom connector, create the replication slots manually.
- Installation and configuration on a Linux machine
- Lessons learned from running Debezium

# Data Migration to Hyperscale Citus

## Setup

Follow all the steps in the setup guide to create an environment to support the following steps.

## Configure Server Parameters

To support the migration, set the source and target PostgreSQL instance parameters to allow for faster egress and ingress. Follow the steps in server-parameters.

## Prerequisites

A database modified to support Citus sharding. To study a sample app that employs Citus features, consult the sample application described in the Citus sample application setup document.

## Schema

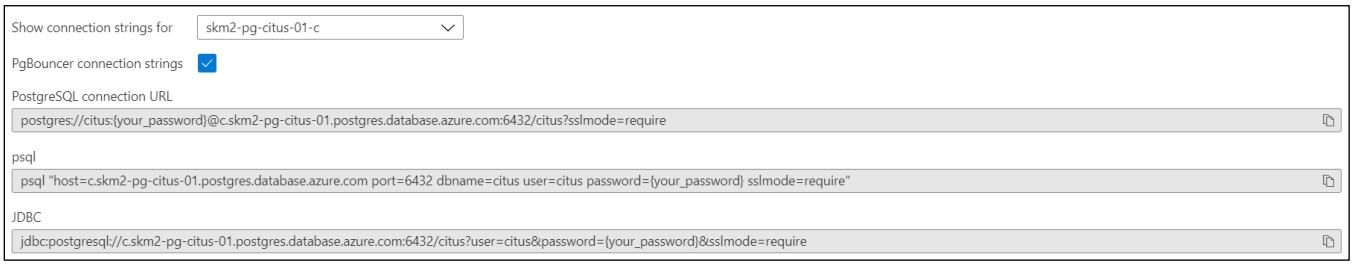
Typically, a COPY-based migration requires the target to contain the migrated schema. Follow the steps in the data-migration---schemakey.

## Data

For larger databases with many tables with small migration windows, parallelize the import using a scripting environment, such as PowerShell, and the COPY command.

Note that for Hyperscale Citus, connections can be made to the coordinator node or any of the worker nodes. Azure Database for PostgreSQL Hyperscale Citus also supports pgBouncer, so it is likely that a connection will be made to the pgBouncer endpoint rather than to any worker nodes directly. In the script below, replace the connection string with the pgBouncer endpoint, which runs on port 6432.

- Open the Azure Portal.
- Browse to the Azure Database for PostgreSQL server group.
- Select **Connection strings**.
- Check the **PgBouncer connection strings** checkbox.
- Copy the connection string details for use in the next step.




Obtaining pgBouncer-equipped connection strings from the Azure portal.

**COPY**

- Switch to the **PREFIX-vm-pgdb01** virtual machine.
- Create a new PowerShell script populated with the code below at c:\temp\migrate_copy.ps1. You can use **Windows PowerShell ISE** for this. Remember to update the local connection information and the pgBouncer target endpoint.

```
function GetData($connString, $sql, $outputFile)
{
   $env:PGPASSWORD = $connString.Password;

   Write-Host "$global:psqlPath\psql.exe -h $($connString.Host) -p $($connString.Port) -U $($connString.User) -d
$($connstring.DbName) -c `"$sql;`"";

   if ($connString.useSSL)
   {
      $env:PGSSLMODE = "require";
   }

   $data = $sql | & $global:psqlPath\psql.exe -h $($connString.Host) -p $($connString.Port) -U $($connString.Us
er) -d $($connstring.DbName) -c `"$sql`";

   return $data;
}

function  ExportAllTables($source)
{
   #get the list of tables...
   $sql = "select table_name, table_schema from information_schema.tables where table_type = 'BASE TABLE' and
table_schema = '$($source.schemaname)' order by table_name;";
   $data = GetData $source $sql $outputFile;

   foreach($item in $data)
   {
```

```
        if ($item.contains("table_name") -or $item.contains("rows)") -or $item.contains("-+-"))
        {
            continue;
        }

        $tableName, $schemaName = $item.split('|');

        if ( $tableName)
        {
            $tableName = $tableName.replace(" ","");
            $schemaName = $schemaName.replace(" ","");
            $fullName = $schemaName + "." + $tablename;

            mkdir "c:/export/$schemaName" -ea SilentlyContinue;

            $sql = "COPY (SELECT * FROM $fullName) TO 'c:/export/$schemaName/$($tableName).copy';";

            #export via psql...
            $data = GetData $source $sql $outputFile;
        }
    }
}

function ImportAllTables($source, $target)
{
    #get the list of tables...
    $sql = "select table_name, table_schema from information_schema.tables where table_type = 'BASE TABLE' and
table_schema = '$($source.schemaname)' order by table_name;";
    $data = GetData $source $sql $outputFile;

    foreach($item in $data)
    {
        if ($item.contains("table_name") -or $item.contains("rows)") -or $item.contains("-+-"))
        {
            continue;
        }

        $tableName, $schemaName = $item.split('|');

        if ( $tableName)
        {
            $tableName = $tableName.replace(" ","");
            $schemaName = $schemaName.replace(" ","");
```

```
        $fullName = $schemaName + "." + $tablename;

        #truncate the table first...
        $sql = "truncate table $fullName;";
        $res = GetData $target $sql $outputFile;

        #import the data...
        $sql = "\COPY $fullName FROM 'c:/export/$schemaName/$($tableName).copy';";
        $res = GetData $target $sql $outputFile;
    }
  }
}

$global:psqlPath = "C:\Program Files\PostgreSQL\10\bin";

$sourceConnStringA = @{};
$sourceConnStringA.Host = "localhost";
$sourceConnStringA.Port = "5432";
$sourceConnStringA.User = "postgres";
$sourceConnStringA.Password = "Seattle123";
$sourceConnStringA.DbName = "reg_app_multitenant";
$sourceConnStringA.SchemaName = "reg_app";
$sourceConnStringA.UseSSL = $false;

$targetConnStringA = @{};
$targetConnStringA.Host = "c.{servername}.postgres.database.azure.com";
$targetConnStringA.Port = "6432";
$targetConnStringA.User = "citus";
$targetConnStringA.Password = "Seattle123Seattle123";
$targetConnStringA.DbName = "citus";
$targetConnStringA.UseSSL = $true;
```

- Run the PowerShell script to start the migration.

```
. c:\temp\migrate_copy.ps1

ExportAllTables $sourceConnStringA;

ImportAllTables $sourceConnStringA $targetConnStringA;
```

Here is a sample output of running the ExportAllTables function. It writes the contents of all tables in the specified source schema to individual files.

```
PS C:\Users\s2admin> ExportAllTables $sourceConnStringA;
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "select t
able_name, table_schema from information_schema.tables whe
re table_type = 'BASE TABLE' and table_schema = 'reg_app' order by table_name;;"


    Directory: C:\export


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        6/20/2021   3:34 AM                reg_app

C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.attendees) TO 'c:/export/reg_app/atten
dees.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.attendees_audit) TO 'c:/export/reg_app
/attendees_audit.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.companies) TO 'c:/export/reg_app/compa
nies.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.ddl_history) TO 'c:/export/reg_app/ddl
_history.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.events) TO 'c:/export/reg_app/events.c
opy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.jobs) TO 'c:/export/reg_app/jobs.copy'
;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.registrations) TO 'c:/export/reg_app/r
egistrations.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.sessions) TO 'c:/export/reg_app/sessio
ns.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "COPY (
SELECT * FROM reg_app.speakers) TO 'c:/export/reg_app/speake
rs.copy';;"
```

Here is the result of invoking the ImportAllTables function. It truncates target tables and then uses client-side \COPY to place data in the target tables.

```
PS C:\Users\s2admin> ImportAllTables $sourceConnStringA $targetConnStringA;

C:\Program Files\PostgreSQL\10\bin\psql.exe -h localhost -p 5432 -U postgres -d reg_app_multitenant -c "select t
able_name, table_schema from information_schema.tables whe
re table_type = 'BASE TABLE' and table_schema = 'reg_app' order by table_name;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.attendees;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.attendees FROM 'c:/export/reg_ap
p/attendees.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.attendees_audit;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.attendees_audit FROM 'c:/export/
reg_app/attendees_audit.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.companies;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.companies FROM 'c:/export/reg_ap
p/companies.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.ddl_history;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.ddl_history FROM 'c:/export/reg_
app/ddl_history.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.events;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.events FROM 'c:/export/reg_app/e
vents.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.jobs;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.jobs FROM 'c:/export/reg_app/job
s.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.registrations;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.registrations FROM 'c:/export/re
g_app/registrations.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.sessions;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
```

```
-d citus -c "\COPY reg_app.sessions FROM 'c:/export/reg_app
/sessions.copy';;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "truncate table reg_app.speakers;;"
C:\Program Files\PostgreSQL\10\bin\psql.exe -h c.skm2-pg-citus-01.postgres.database.azure.com -p 6432 -U citus
-d citus -c "\COPY reg_app.speakers FROM 'c:/export/reg_app
/speakers.copy';;"
```

## Revert Server Parameters

With the migration completed, revert the server parameters to support the workload. Follow the steps in revert-server-parameters.

### Enable Keys and Triggers

As the final step, run the SQL to reset all foreign keys and to enable all triggers.

# Data Migration

## Back up the database

Lower the risk and back up the database before upgrading or migrating data. Use the PostgreSQL pgAdmin UI or the pg_dump command to export the database for restore capability.

## Offline vs. Online

Before selecting a migration tool, decide if the migration should be online or offline.

- **Offline migrations** require the system to be down while the migration takes place. Users will not be able to modify data. This option ensures that the state of the data will be exactly what is expected when restored in Azure.
- **Online migrations** will migrate the data in near real-time. This option is appropriate when there is little downtime for the users or application consuming the data workload. The costs are too high for the corporation to wait for a complete migration. The process involves replicating the data using a replication method such as logical replication or similar functionality.

**Case Study:** In the case of WWI, their environment has some complex networking and security requirements that will not allow for the appropriate changes to be applied for inbound and outbound connectivity in the target migration time frame. These complexities and requirements essentially eliminate the online approach from consideration.

**Note:** Review the Planning and Assessment sections for more details on Offline vs Online migration.

## Data Drift

Offline migration strategies have the potential for data drift. Data drift occurs when newly modified source data becomes out of sync with migrated data. When this happens, a full export or a delta export is necessary. To mitigate this problem, stop all traffic to the database and then perform the export. If stopping all data modification traffic is not possible, it will be necessary to account for the data drift.

Determining the changes can become complicated if the database tables do not have columns such as numeric primary keys, or some type of modification and creation date in every table that needs to be migrated.

For example, if a numeric primary key is present and the migration is importing in sort order, it will be relatively simple to determine where the import stopped and restart it from that position. If no numeric key is present, then utilize modification and creation date, and again, import in a sorted manner to restart the migration from the last timestamp seen in the target.

# Performance recommendations

## Unlogged tables

Unlogged tables are a PostgreSQL feature that optimizes bulk inserts. PostgreSQL uses Write-Ahead Logging (WAL). It provides atomicity and durability by default. Atomicity, consistency, isolation, and durability make up the ACID properties.

Inserting into an unlogged table means that PostgreSQL executes inserts without writing into the transaction log, which is an I/O operation. As a result, these tables are considerably faster than ordinary tables.

Use the following commands to configure unlogged tables:

```
CREATE UNLOGGED TABLE <tableName>
```

OR

```
ALTER TABLE <tableName> SET UNLOGGED
```

Use the following commands to resume logging.

```
ALTER TABLE <tableName> SET LOGGED
```

## Connection Pooling

In PostgreSQL, establishing a database connection is an expensive operation because each new connection to the PostgreSQL instance requires forking the OS process and allocating memory for the connection. As a result, transactional applications that frequently open and close connections at the end of transactions can experience higher connection latency, resulting in lower database throughput (transactions per second) and overall higher application latency.

Migration strategies requiring many parallel threads should utilize connection pooling tools such as PgBouncer or Pgpool.

These tools enable faster data migration because there no need to create and tear down database connections.

PgBouncer can improve the data migration throughput by almost 4x times and database connectivity latency by a factor of 40.

### PgBouncer

PgBouncer is the preferred method of doing connection pooling and is supported with Azure Database for PostgreSQL.

PgBouncer has these additional benefits:

- Improved throughput and performance
- No connection leaks by defining the maximum number of connections to the database server

- Improved connection resiliency against restarts
- Reduced memory fragmentation

PgBouncer, with its built-in retry logic, can also ensure connection resiliency, high availability, and transparent application failover during planned (scheduled/scale-up/scale-down) or unplanned database server failover.

PgBouncer is recommended for general application usage. Applications that do not implement connection pooling can leverage PgBouncer as a connection pooling proxy and can benefit from the performance gains.

### Source Tool Network

When running the migration tool on a virtual machine, it is possible to change the TCP_NODELAY setting. By default, TCP uses Nagle's algorithm, which optimizes by batching outgoing packets. The result is fewer sent packets. This strategy works well if the application sends packets frequently and latency is not the highest priority. Realize latency improvements by sending on sockets created with the TCP_NODELAY option enabled. This results in lower latency but more sends. Consider this client-side setting for the Virtual Machine (VM). Applications that benefit from the TCP_NODELAY option typically tend to **do smaller, infrequent writes and are particularly sensitive to latency**. As an example, alter this setting to reduce latency from 15-40 ms to 2-3 ms.

To change this setting on Windows machines, do the following:

- Open the **REGEDIT** tool
- Under the subtree HKEY_LOCAL_MACHINE, find the SYSTEM\CurrentControlSet\services\Tcpip\Parameters\Interfaces key
- Find the correct network interface
- In the space, right-click and select **New** for creating a DWORD Value
- For the value name, type **TcpNoDelay**
- For the Dword value, type **1**
  - In the empty space, right-click, and select **New** for creating a DWORD Value
- For the value name, type **TcpAckFrequency**
- For the Dword value, type **1**
- Close the REGEDIT tool

### Exporting

- Upgrade the data and log disks, if experiencing poor export performance.
- Use an export tool that leverages multiple threads.

- When using PostgreSQL 9.6 or higher, use partitioned tables when appropriate to increase the speed of exports.
- When logging is enabled, move the log files directory to a separate drive than the data drive.

### Importing

- Create clustered indexes and primary keys **after** loading data.
- Load data in primary key order or according to some date column (such as modification date or creation date).
- Delay the creation of secondary indexes until after data is loaded. Create all secondary indexes after loading.
- Disable foreign key constraints before loading. Disabling foreign key checks provides significant performance gains. Enable the constraints and verify the data after the load to ensure referential integrity.
- Load data in parallel.
  - **Warning:** Too much parallelism can cause resource contention. Monitor resources by using the metrics available in the Azure portal.
- Use multi-valued INSERT commands to decrease the overhead of many single inserts.
- Modify the WAL log file location to put the WAL log files on a separate drive to gain extra performance.
- Set shared_buffers to at least 25% of available memory.
- Increase the wal_buffers server parameter to a higher value when performing parallel loading that will initiate many connections.

**Note**: The PostgreSQL wiki has several articles focused on performance.

### Post Import

- Run the ANALYZE command to update all statistics.

## Performing the Migration Checklist

- Back up the database
- Create and verify the Azure Landing zone
- Export and configure Source Server Parameters
- Export and configure Target Server Parameters
- Export the database objects (schema, users, etc.)

- Export the data
- Import the database objects
- Import the data (no triggers, keys)
- Validate the data
- Configure triggers and keys
- Reset Source Server Parameters
- Reset Target Server Parameters
- Migrate the Application(s)

## Common Steps

Despite the path taken, there are common steps in the process:

- Upgrade to a supported Azure PostgreSQL version
- Inventory database objects
- Export users and permissions

## Migrate to the latest PostgreSQL version

The WWI Conference database, which currently runs 9.5, will be upgraded to PostgreSQL 11.0, the latest version supported by Azure Database for PostgreSQL.

There are two tool options for upgrading to 11.0:

- In-place with pg_upgrade
- Export/Import with pg_dumpall

## WWI Use Case

After successfully migrating the PostgreSQL instance to 9.6, the WWI migration team realized the original Database Migration Service (DMS) migration is unavailable to them, as the DMS tool currently only supports 10.0 and higher. DMS also required network access, and the WWI migration team was not ready to handle their complex network issues. These environmental issues narrowed their migration tool choice to PostgreSQL pgAdmin.

## Database Objects

As outlined in the test-plans section, take an inventory of database objects before and after the migration.

Migration teams should develop and test helpful inventory SQL scripts before beginning the migration phase.

Database object inventory SQL examples:

```
CREATE OR REPLACE FUNCTION public.migration_performinventory(
        schema_name character)
    RETURNS void
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
        row_table record;
        cur_tables cursor for select table_name, table_schema from information_schema.tables where table_typ
e = 'BASE TABLE' and table_schema = schema_name order by table_name;
        count int;

begin

CREATE TABLE IF NOT EXISTS MIG_INVENTORY
(
        REPORT_TYPE VARCHAR(1000),
        OBJECT_NAME VARCHAR(1000),
        PARENT_OBJECT_NAME VARCHAR (1000),
        OBJECT_TYPE VARCHAR(1000),
        COUNT INT
);

ALTER TABLE MIG_INVENTORY REPLICA IDENTITY FULL;

        --clear it out...
        delete from mig_inventory;

        --count of tables
        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'TABLES', 'TABLES', COUNT(*)
    FROM
                information_schema.tables
    where
```

```
                TABLE_SCHEMA = schema_Name
                and table_type = 'BASE TABLE';

                --count of stats
                INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'STATISTICS', 'STATISTICS', COUNT(*)
FROM
                PG_INDEXES pg
        WHERE
                pg.schemaname = schema_Name;

                --count of table constraints
                INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'TABLE_CONSTRAINTS', 'TABLE_CONSTRAINTS', COUNT(*)
FROM
                information_schema.table_constraints
        WHERE
                TABLE_SCHEMA = schema_Name;

                INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'VIEWS', 'VIEWS', COUNT(*)
        FROM
                information_schema.VIEWS
        WHERE
                TABLE_SCHEMA = schema_Name;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'FUNCTIONS', 'FUNCTIONS', COUNT(*)
        FROM
                information_schema.ROUTINES
        WHERE
                ROUTINE_TYPE = 'FUNCTION' and
                ROUTINE_SCHEMA = schema_Name;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'PROCEDURES', 'PROCEDURES', COUNT(*)
        FROM
                information_schema.ROUTINES
```

```
        WHERE
                ROUTINE_TYPE = 'PROCEDURE' and
                ROUTINE_SCHEMA = schema_Name;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'USER DEFINED FUNCTIONS', 'USER DEFINED FUNCTIONS', COUNT(*)
        from pg_proc p
        left join pg_namespace n on p.pronamespace = n.oid
        left join pg_language l on p.prolang = l.oid
        left join pg_type t on t.oid = p.prorettype
        where n.nspname not in ('pg_catalog', 'information_schema')      ;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'TRIGGERS', 'TRIGGERS', COUNT(*)
        from information_schema.triggers;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'USERS', 'USERS', COUNT(*)
  FROM
                pg_catalog.pg_user;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'EXTENSIONS', 'EXTENSIONS', COUNT(*)
        FROM pg_extension;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'FDW', 'FDW', COUNT(*)
        FROM pg_catalog.pg_foreign_data_wrapper fdw;

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        SELECT
                'OBJECTCOUNT', 'USER DEFINED TYPES', 'USER DEFINED TYPES', COUNT(*)
        FROM       pg_type t
        LEFT JOIN   pg_catalog.pg_namespace n ON n.oid = t.typnamespace
        WHERE      (t.typrelid = 0 OR (SELECT c.relkind = 'c' FROM pg_catalog.pg_class c WHERE c.oid = t.typrel
id))
        AND     NOT EXISTS(SELECT 1 FROM pg_catalog.pg_type el WHERE el.oid = t.typelem AND el.typarray
= t.oid)
```

```
        AND     n.nspname NOT IN ('pg_catalog', 'information_schema');

        INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, COUNT)
        select
                'OBJECTCOUNT', 'LANGUAGES', 'LANGUAGES', COUNT(*)
        from pg_language;

        for row_table in cur_tables loop

         EXECUTE format('select count(*) from %I.%I', schema_name, row_table.table_name) into count;

         INSERT INTO MIG_INVENTORY (REPORT_TYPE,OBJECT_NAME, OBJECT_TYPE, PARENT_OBJECT_NAME, C
OUNT)
                SELECT
                        'TABLECOUNT', row_table.table_name, 'TABLECOUNT', schema_name, count;

        end loop;

end;
$BODY$;

select Migration_PerformInventory('reg_app');

select * from mig_inventory;
```

- Calling this procedure on the source DB reveals the following:

| | report_type character varying (1000) | object_name character varying (1000) | parent_object_name character varying (1000) | object_type character varying (1000) | count integer |
|---|---|---|---|---|---|
| 1 | OBJECTCOUNT | TABLES | [null] | TABLES | 8 |
| 2 | OBJECTCOUNT | STATISTICS | [null] | STATISTICS | 9 |
| 3 | OBJECTCOUNT | TABLE_CONSTRAINTS | [null] | TABLE_CONSTRAINTS | 32 |
| 4 | OBJECTCOUNT | VIEWS | [null] | VIEWS | 2 |
| 5 | OBJECTCOUNT | FUNCTIONS | [null] | FUNCTIONS | 3 |
| 6 | OBJECTCOUNT | PROCEDURES | [null] | PROCEDURES | 0 |
| 7 | OBJECTCOUNT | USER DEFINED FUNCTIONS | [null] | USER DEFINED FUNCTIONS | 15 |
| 8 | OBJECTCOUNT | TRIGGERS | [null] | TRIGGERS | 2 |
| 9 | OBJECTCOUNT | USERS | [null] | USERS | 5 |
| 10 | OBJECTCOUNT | EXTENSIONS | [null] | EXTENSIONS | 5 |
| 11 | OBJECTCOUNT | FDW | [null] | FDW | 1 |
| 12 | OBJECTCOUNT | USER DEFINED TYPES | [null] | USER DEFINED TYPES | 2 |
| 13 | OBJECTCOUNT | LANGUAGES | [null] | LANGUAGES | 4 |
| 14 | TABLECOUNT | attendees | reg_app | TABLECOUNT | 100 |
| 15 | TABLECOUNT | attendees_audit | reg_app | TABLECOUNT | 0 |
| 16 | TABLECOUNT | ddl_history | reg_app | TABLECOUNT | 19 |
| 17 | TABLECOUNT | events | reg_app | TABLECOUNT | 1 |
| 18 | TABLECOUNT | jobs | reg_app | TABLECOUNT | 0 |
| 19 | TABLECOUNT | registrations | reg_app | TABLECOUNT | 50 |
| 20 | TABLECOUNT | sessions | reg_app | TABLECOUNT | 4 |
| 21 | TABLECOUNT | speakers | reg_app | TABLECOUNT | 4 |

The output from the database migration inventory function

## Execute migration

With the basic migration components in place, it is now possible to proceed with the data migration. WWI will utilize the PostgreSQL pgAdmin option to export the data and then import it into Azure Database for PostgreSQL.

Options:

- Back up the database
- Copy command
- Azure Data Factory (ADF)
- Azure Database Migration Service (DMS)
- Logical Replication
- Logical Decoding
- Non-Citus to Citus

Once the data is migrated, point the application to the new instance

- Migrate Application Settings

Lastly, validate the target database's inventory. Below is an example of the SQL results in a target environment. It is easy to identify object count discrepancies.

**Note:** Unsupported objects will be dropped during the migrations. Certain choices were also made during the migration that resulted in discrepancies.

| | report_type character varying (1000) | object_name character varying (1000) | parent_object_name character varying (1000) | object_type character varying (1000) | count integer |
|---|---|---|---|---|---|
| 1 | OBJECTCOUNT | TABLES | [null] | TABLES | 8 |
| 2 | OBJECTCOUNT | STATISTICS | [null] | STATISTICS | 9 |
| 3 | OBJECTCOUNT | TABLE_CONSTRAINTS | [null] | TABLE_CONSTRAINTS | 32 |
| 4 | OBJECTCOUNT | VIEWS | [null] | VIEWS | 2 |
| 5 | OBJECTCOUNT | FUNCTIONS | [null] | FUNCTIONS | 3 |
| 6 | OBJECTCOUNT | PROCEDURES | [null] | PROCEDURES | 0 |
| 7 | OBJECTCOUNT | USER DEFINED FUNCTIONS | [null] | USER DEFINED FUNCTIONS | 15 |
| 8 | OBJECTCOUNT | TRIGGERS | [null] | TRIGGERS | 2 |
| 9 | OBJECTCOUNT | USERS | [null] | USERS | 5 |
| 10 | OBJECTCOUNT | EXTENSIONS | [null] | EXTENSIONS | 5 |
| 11 | OBJECTCOUNT | FDW | [null] | FDW | 1 |
| 12 | OBJECTCOUNT | USER DEFINED TYPES | [null] | USER DEFINED TYPES | 2 |
| 13 | OBJECTCOUNT | LANGUAGES | [null] | LANGUAGES | 4 |
| 14 | TABLECOUNT | attendees | reg_app | TABLECOUNT | 100 |
| 15 | TABLECOUNT | attendees_audit | reg_app | TABLECOUNT | 0 |
| 16 | TABLECOUNT | ddl_history | reg_app | TABLECOUNT | 19 |
| 17 | TABLECOUNT | events | reg_app | TABLECOUNT | 1 |
| 18 | TABLECOUNT | jobs | reg_app | TABLECOUNT | 0 |
| 19 | TABLECOUNT | registrations | reg_app | TABLECOUNT | 50 |
| 20 | TABLECOUNT | sessions | reg_app | TABLECOUNT | 4 |
| 21 | TABLECOUNT | speakers | reg_app | TABLECOUNT | 4 |

Running the migration tool against the migrated PostgreSQL instance.

## Data Migration Checklist

- Understand the complexity of the environment and determine if an online approach is feasible.
- Account for data drift. Stopping the database service can eliminate potential data drift. Acceptable downtime costs?
- Configure source server parameters for fast export.
- Configure target server parameters for fast import.
- Test any migrations that have a different source version different from the target version.
- Migrate any miscellaneous objects, such as usernames and privileges.
- Disable foreign keys and triggers before the import. Re-enable after the import.
- Update application settings to point to the new instance.
- Document all tasks. Update task completion to indicate progress.

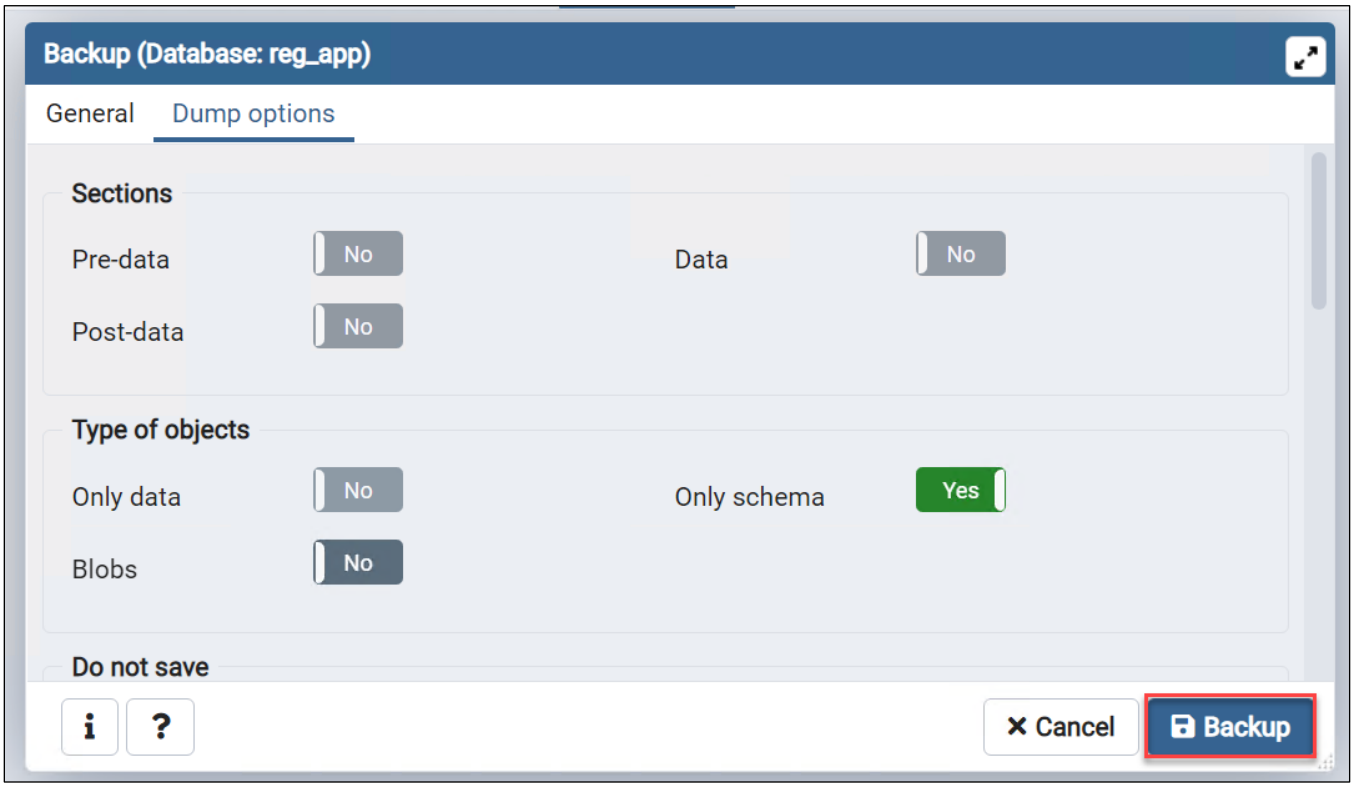
# Data Migration - Schema

## Schema Migration

As most tools require schema migration prior to their use, this document provides the steps to export and import the schema.

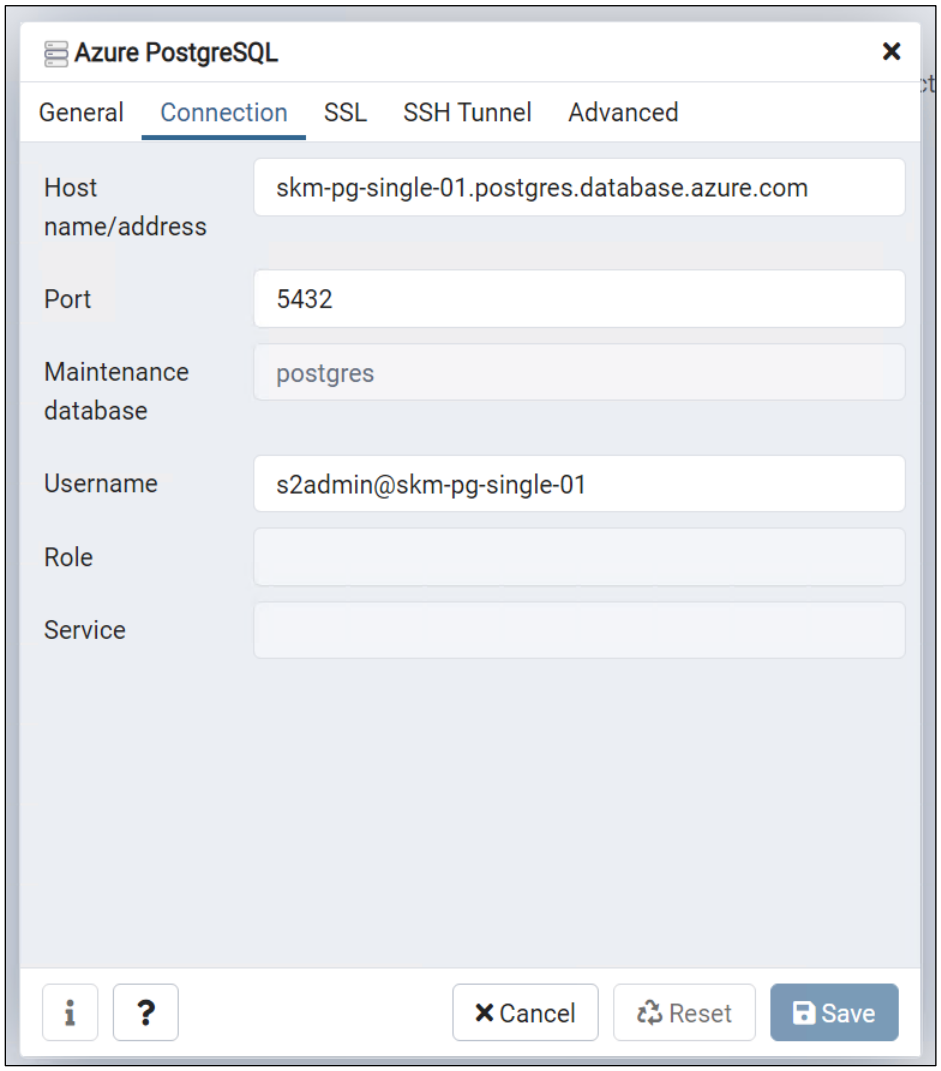
### Export the schema

- Open PostgreSQL pgAdmin and connect as the local database's postgres user.
- Expand the **Databases** node
- Select the **reg_app** database. However, if you are performing the Citus migration, select **reg_app_multitenant**.
- In the application menu, select **Tools->Backup**
- For the file name, type C:\temp\reg_app_schema.sql.
- For the format, select **Plain**
- For the encoding, select **UTF8**
- Select the **Dump options** tab
- Toggle the following options
  - **Only schema** option to **Yes**
  - All other options to **No**
- Select **Backup**




Using the pgAdmin tool to export the on-premises database schema.

- If it does not exist, create a new connection to the Azure Database for PostgreSQL in pgAdmin.
  - For Name, enter a name such as **Azure PostgreSQL**
  - For Hostname, enter the full server DNS (ex: servername.postgresql.database.azure.com).
  - Enter the username (ex: s2admin@servername).
  - Enter the password (**Seattle123Seattle123**).
  - Select the **SSL** tab.
  - For the Root certificate, browse to the **BaltimoreCyberTrustRoot.crt.pem** key file.
  - Select **Save**.




PostgreSQL connection dialog box is displayed.

- If it does not exist, create the reg_app database. Note that you cannot do this with Azure Database for PostgreSQL (Citus) – you must use the default citus database.
  - Expand the **Databases** node.
  - Right-click the **Databases** node and select **Create->Database**.
  - For the name, type **reg_app**
  - Select **Save**.
- Right-click the **reg_app** database and select **Query Tool**.
- Browse to the c:\temp\reg_app_schema.sql script.

- Press **F5** to run the script. Observe the various errors returned.

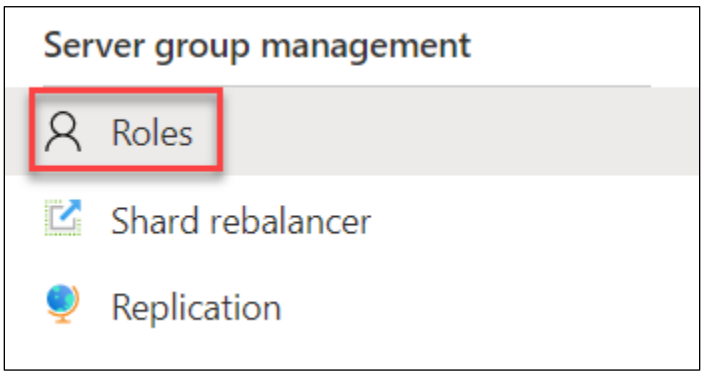
### Export Users and Roles

The first error encountered will be focused on users and roles. We must export that data from the source and import it into the target environment. On the source system, perform the following:

- Using the query tool connected to the source, run the following query. Then, execute c:\temp\server-users.sql on the target database.

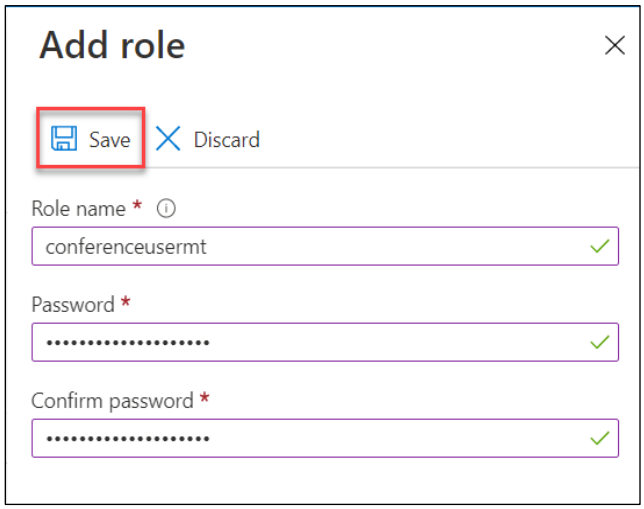
```
 COPY(
SELECT 'CREATE ROLE ' || usename || ' WITH LOGIN PASSWORD ''Seattle123Seattle123'';' as cmd
FROM pg_catalog.pg_user
 )
 TO 'c:\temp\server-users.sql';
```

- With Hyperscale Citus, the citus user cannot create roles. You can use the Azure Portal to add more users. Azure will then propagate this change to all nodes in the cluster.
  - Navigate to your Citus instance in the Azure portal, and below **Server group management**, select **Roles**.




Adding new Roles to the Citus cluster using the Azure portal.

  - Select **+ Add**. Enter the **Role name** and **Password**. Then, select **Save**.




Creating a new conferenceusermt role in the Azure portal for the multi-tenant app.

- Since you are logged in as the s2admin user, add the user to the appropriate roles you just imported. Be cautious that there is already a superuser role called postgres in Citus instances.

```
GRANT postgres TO s2admin;
```

- In addition to the users, you would need to assign the server level roles and permissions. If you are using Citus, creating roles in the Azure Portal automatically assigns the LOGIN privilege.

```
 COPY(
  SELECT
   'ALTER USER ' || usename || ' ' ||
   CASE
    WHEN usesuper AND usecreatedb THEN
     CAST('superuser CREATEDB' AS pg_catalog.text)
    WHEN usesuper THEN
     CAST('superuser' AS pg_catalog.text)
    WHEN usecreatedb THEN
     CAST('CREATEDB' AS pg_catalog.text)
    ELSE
     CAST('' AS pg_catalog.text)
   END || ';' as cmd
  FROM pg_catalog.pg_user
)
TO 'c:\temp\server-level-roles.sql';
```

- Run c:\temp\server-level-roles.sql in the Azure Database for PostgreSQL command window

**Note**: The superuser role cannot be assigned in Azure Database for PostgreSQL. However, you can assign the azure_pg_admin role (e.g. GRANT azure_pg_admin TO postgres) in non-Citus offerings.

### Removing unsupported features

- Run the script repeatedly. Errors will be displayed. Remove the unsupported features until the script runs successfully without errors. Some hints:
- Remove the addition of procedural languages (plperl).
- Remove the addition of the unsupported extensions (plpython3u, file_fdw).

  **Note**: Browse which extensions are supported by Azure Database for PostgreSQL using SELECT * FROM pg_available_extensions;.
- Remove objects that use the above items (perl_max, pymax).
- Remove other objects (**CAST with reference to pymax**).
- Remove any Foreign Data Wrappers (file_fdw).

For some Azure Database for PostgreSQL service types, it will be necessary to remove the following:

- All CREATE TEXT SEARCH items

**Note**: In case of permissions error, it may be easier to replace all owner statements to point to the s2admin user or in the case of Citus, the citus user.

**Note**: If you experience any issues with Event Triggers, remove the ALTER EVENT TRIGGER ... OWNER TO ... SQL statements. If that still does not resolve the issue, remove the event triggers and their underlying functions.

### Set tables to unlogged

Run the following query to create a script to set all tables to unlogged. Replace the schema_name placeholder:

```
COPY(
select 'ALTER TABLE ' || table_name || ' SET UNLOGGED;' as cmd
from information_schema.tables
where table_type = 'BASE TABLE'
and table_schema = {schema_name}
order by table_name
)
  TO 'c:\temp\set-tables-unlogged.sql';
```

Run the following query to create a script to set all tables back to logged:

```
COPY(
select 'ALTER TABLE ' || table_name || ' SET LOGGED;' as cmd
from information_schema.tables
```

```
where table_type = 'BASE TABLE'
and table_schema = {schema_name}
order by table_name
)
  TO 'c:\temp\set-tables-logged.sql';
```

On the target, run the query to set the tables to unlogged.

**Warning**: This should be done to improve insert performance, but if there are failures during the process, possible data loss could occur, and the migration would need to be restarted. Be sure to verify the inventory of the target once the migration is completed.

### Remove Keys and Triggers

If you are performing a Citus migration, we recommend performing the table distribution after you disable foreign keys, and before you enable them. The distribution functions do not need to be called in a particular order.

- Open Command Prompt
- Connect to the Azure Database for PostgreSQL using the following command. Replace the user and servername tokens. The servername token is not required for a migration to Flexible Server. If you are using Citus, use --dbname=citus and --username=citus.

```
psql --host={servername}.postgres.database.azure.com --port=5432 --username={user}@{servername} --dbname=reg_app
```

- Paste the following code into the PSQL prompt. It will create a temporary view containing SQL statements. The PSQL \COPY command will write the contents of the view to a file, simplifying the migration experience. You may encounter issues using these scripts to record composite foreign keys. Try other utilities..

```
CREATE TEMP VIEW ForeignKeys AS
SELECT
      concat('alter table ',  Queries.table_schema, '.', Queries.tablename, ' ', STRING_AGG(concat('DROP CONSTRAIN
T ', Queries.foreignkey), ','), ';') as DropQuery
   FROM
   (SELECT
   tc.table_schema,
   tc.constraint_name as foreignkey,
   tc.table_name as tableName,
   kcu.column_name,
   ccu.table_schema AS foreign_table_schema,
   ccu.table_name AS foreign_table_name,
   ccu.column_name AS foreign_column_name
FROM
   information_schema.table_constraints AS tc
```

```
  JOIN information_schema.key_column_usage AS kcu
    ON tc.constraint_name = kcu.constraint_name
    AND tc.table_schema = kcu.table_schema
  JOIN information_schema.constraint_column_usage AS ccu
    ON ccu.constraint_name = tc.constraint_name
    AND ccu.table_schema = tc.table_schema
WHERE constraint_type = 'FOREIGN KEY') Queries
 GROUP BY Queries.table_schema,Queries.tablename;
```

- \COPY the query results to a new file. You can use the server-side COPY statement (shown below) if you create the temporary views on the source PostgreSQL instance. However, if you create the temporary views on the target, use the client-side \COPY (the second command).

```
COPY (select * from ForeignKeys)
TO 'c:\temp\remove_constraints.sql';

\copy (SELECT * FROM ForeignKeys)
TO 'c:\temp\remove_constraints.sql'
```

- Repeat the same process to create the add foreign keys script:

```
CREATE TEMP VIEW AddForeignKeys AS
SELECT
      concat('alter table ', Queries.table_schema, '.', Queries.tablename, ' ',
      STRING_AGG(concat('ADD CONSTRAINT ', Queries.foreignkey, ' FOREIGN KEY (', column_name, ')', 'REFERENC
ES ', foreign_table_name, '(', foreign_column_name, ')' ), ','), ';') as AddQuery
  FROM
  (SELECT
  tc.table_schema,
  tc.constraint_name as foreignkey,
  tc.table_name as tableName,
  kcu.column_name,
  ccu.table_schema AS foreign_table_schema,
  ccu.table_name AS foreign_table_name,
  ccu.column_name AS foreign_column_name
FROM
  information_schema.table_constraints AS tc
  JOIN information_schema.key_column_usage AS kcu
    ON tc.constraint_name = kcu.constraint_name
    AND tc.table_schema = kcu.table_schema
  JOIN information_schema.constraint_column_usage AS ccu
    ON ccu.constraint_name = tc.constraint_name
    AND ccu.table_schema = tc.table_schema
```

```
WHERE constraint_type = 'FOREIGN KEY') Queries
 GROUP BY Queries.table_schema,Queries.tablename;
```

- Copy the statements into a SQL file. Then, press Ctrl+C in the PSQL interpreter to close the connection. Remember, you have the option of using server-side or client-side COPY; it depends on whether you create the temporary views on the source or target.

```
COPY (select * from AddForeignKeys)
TO 'c:\temp\add_constraints.sql';

\copy (SELECT * FROM AddForeignKeys)
TO 'c:\temp\add_constraints.sql'
```

- Use PSQL to execute the remove_constraints.sql file. This needs to be executed on the target instance.

```
psql --host=[PREFIX]-pg-single-01.postgres.database.azure.com --port=5432 --username=postgres@[PREFIX]-pg-single-01 --dbname=reg_app --file c:\temp\remove_constraints.sql
```

**Note**: This will drop four foreign keys for the single-tenant migration path shown in this document.

- Open another PSQL connection to the Azure instance. Create another temporary view with commands to disable triggers.

```
CREATE TEMP VIEW DisableTriggers AS
select concat ('alter table ', event_object_schema, '.',event_object_table, ' disable trigger ', trigger_name)
from information_schema.triggers;
```

- Write the contents of a view to another SQL file

```
COPY (select * from DisableTriggers)
TO 'c:\temp\disable_triggers.sql';

\copy (SELECT * FROM DisableTriggers)
TO 'c:\temp\disable_triggers.sql'
```

- Create a temporary view that contains the trigger enabling scripts

```
CREATE TEMP VIEW EnableTriggers AS
select concat ('alter table ', event_object_schema, '.',event_object_table, ' enable trigger ', trigger_name)
from information_schema.triggers;
```

- Write the contents of the view to another SQL file

```
COPY (select * from EnableTriggers)
TO 'c:\temp\enable_triggers.sql';
```

```
\copy (SELECT * FROM EnableTriggers)
TO 'c:\temp\enable_triggers.sql'
```

- Execute the disable_triggers.sql script. This must be executed on the target.

```
psql --host={servername}.postgres.database.azure.com --port=5432 --username={user}@{servername} --dbname=reg_app --file c:\temp\disable_triggers.sql
```

**Note**: This will disable one trigger for the attendees table in the single-tenant migration path.

After running through the previous series of steps, the migration of the supported source schema will be completed.

# Data Migration - Server Parameters - Ingress

## Configuring Server Parameters (Source)

Depending on the type of migration path selected (offline vs. online), evaluate if server parameters will need to be modified to support a fast egress of the data. **Online** migrations may not require server parameter adjustments. Most likely, the migration is performing logical replication and the data is syncing on its own. When performing an **offline** migration, once application traffic is stopped, it is possible to switch the server parameters from supporting the workload to supporting the export.

## Configuring Server Parameters (Target)

Review the server parameters before starting the import process into Azure Database for PostgreSQL. Server parameters can be retrieved and set using the Azure Portal or by calling the Azure CLI to make the changes.

## Exporting Server Parameters

For both the source and target instances, take a backup copy of the server parameters before making any modifications. This step will allow for the configuration of the target to the source instance's workload specifications, after the migration has completed.

### Exporting Server Parameters (PSQL)

- Open a windows command prompt
- Run the following, be sure to update the tokens:

```
psql -h servername -p 5432 -d postgres -U postgres -c "select name,setting,unit,enumvals from pg_settings" -o "c:\temp\settings_postgres.txt"
```

- When prompted, type Seattle123 for the password.

In the new c:\temp\settings_postgres.txt file, review the default PostgreSQL server parameters as shown in Appendix C: Default server parameters PostgreSQL 10.0.

### Exporting Server Parameters (Azure)

Execute the following PowerShell script to export parameters from Azure:

```
$rgName = "{RESOURCE_GROUP_NAME}";
$serverName = "{SERVER_NAME}";

az postgres server configuration list --resource-group $rgName --server $serverName
```

Here is the truncated output.

```
PS C:\Users\s2admin> az postgres server configuration list --resource-group $rgName --server $serverName
[
  {
    "allowedValues": "on,off",
    "dataType": "Boolean",
    "defaultValue": "on",
    "description": "Enable input of NULL elements in arrays.",
    "id": "/subscriptions/30fc406c-c745-44f0-be2d-63b1c860cde0/resourceGroups/[redacted]/providers/Microsoft.DBforPostgreSQL/servers/skm-pg-single-01/configurations/array_nulls",
    "name": "array_nulls",
    "resourceGroup": "sai-postgresql-migration",
    "source": "system-default",
    "type": "Microsoft.DBforPostgreSQL/servers/configurations",
    "value": "on"
  },
```

Obtaining PostgreSQL server parameters using the Azure CLI.

**Note**: Authenticate with the Azure CLI and set the correct subscription to run the command above. Run the following commands to do so:

```
az login
az account set -s [YOUR AZURE SUBSCRIPTION ID]
```

- To run the same commands with the psql tool, download the CA root certification to c:\temp (make this directory).

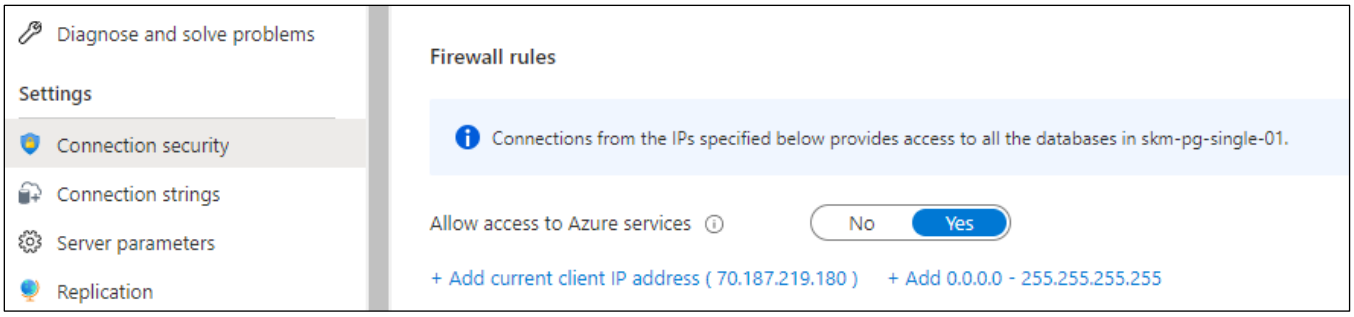
**Note**: The certificate is subject to change. Reference Configure SSL connectivity in your application to securely connect to Azure Database for PostgreSQL for the latest certificate information.

- Run the following in a command prompt, be sure to update the tokens:

```
psql -h [YOUR PREFIX]-pg-single-01.postgres.database.azure.com -p 5432 -d postgres -U s2admin@[YOUR PREFIX]-pg-single-01 -c "select name,setting,unit,enumvals from pg_settings" -o "c:\temp\settings_azure.txt"
```

- When prompted, type Seattle123Seattle123 for the password.

  **Note**: If any errors are displayed, navigate to the database resource in Azure. Then, under **Connection security**, ensure that **Allow access to Azure services** is selected.




  Allowing connections to Azure Database for PostgreSQL from other Azure resources.

In the new settings_azure.txt file, notice the default Azure Database for PostgreSQL server parameters as shown in Appendix D: Default server parameters Azure (Single Server) V11.

## Configure Egress Server Parameters (Source)

The following source server parameters should be set before starting the data export on the source:

- work_mem: 8MB.

**Note**: Sorting of data export will likely occur such that the ingress completes faster at the target. Doubling the default value can potentially provide performance increases at source export.

- maintenance_work_mem: 64MB.

**Note**: This value isn't really used during an export, the default can be used.

- temp_buffers: 1GB.

**Note**: When doing any type of temporary tables when performing the export, you may want to increase this value.

- max_connections: 100.

**Note**: When running a tool that will create over 100 concurrent threads, you will want to increase this value.

When working with logical replication and logical decoding, configure the following parameters:

- wal_level: Logical.
- max_replication_slots: 10.
- max_wal_senders: 10.
- shared_buffers - 128MB

**Note**: When executing multiple threads, more memory will be needed for connections, operations and other items. Give PostgreSQL as much memory as possible during the export operations, yet leaving room for the operating system.

- wal_buffers: -1.

**Note**: This value default is -1 which equates to 3% of the shared_buffers value. Increasing shared_buffers will increase this value, unless you want to allocate more to the wal_buffers.

- max_wal_size: 1024MB.
- min_wal_size: 80MB.

These settings can be updated using the sql statements below:

```
ALTER SYSTEM SET work_mem = 8192;
ALTER SYSTEM SET maintenance_work_mem = 65536;
ALTER SYSTEM SET temp_buffers = 1024;
```

```
ALTER SYSTEM SET max_connections = 100;

ALTER SYSTEM SET wal_level = Logical;
ALTER SYSTEM SET max_replication_slots = 5;
ALTER SYSTEM SET max_wal_senders = 10;

ALTER SYSTEM SET wal_buffers = -1;
ALTER SYSTEM SET max_wal_size = 1024;
ALTER SYSTEM SET min_wal_size = 80;
```

**Note**:

- When executed in pgAdmin, execute each statement individually, as the ALTER SYSTEM command cannot be executed in a transaction block.
- On 32bit source systems, any values to do with memory can not be greater than 2GB.

## Configure Ingress Server Parameters (Target)

The following source server parameters should be set before starting the data migration on the target:

- work_mem: 4096KB.

**Note**: Sorting should already be completed on the source export.

- maintenance_work_mem: 128MB.

**Note**: Larger settings might improve performance for vacuuming and for restoring database dumps, specifically around index creation.

- max_connections: 100.

**Note**: When performing the import and executing many items in parallel, possibly increase this value to accommodate the threads.

- shared_buffers: 256MB

**Note**: When executing multiple threads, there will be a need for more memory for connections, operations, and other items. Give PostgreSQL as much memory as possible during the export operations, yet leaving room for the operating system.

- wal_buffers: 8192

**Note**: Increasing this value can improve write performance on a busy server where many clients are committing at once, as in the case with a migration.

- max_wal_size: 5120MB.

**Note**: Increasing this during an initial migration can speed up the migration.

- min_wal_size: 2560MB.

These settings can be updated using the Azure PowerShell cmdlets below:

```
[Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls
[Net.ServicePointManager]::SecurityProtocol = "tls12, tls11, tls"

$pp = Get-PackageProvider -Name NuGet -Force

Set-PSRepository PSGallery -InstallationPolicy Trusted

Install-Module -Name Az.PostgreSQL

$rgName = "{RESOURCE_GROUP_NAME}";
$serverName = "{SERVER_NAME}";

Select-AzSubscription -Subscription "{SUBSCRIPTION_ID}"

Update-AzPostgreSQLConfiguration -Name "work_mem" -ResourceGroupName $rgname -ServerName $serverNa
me -Value 4096
Update-AzPostgreSQLConfiguration -Name "maintenance_work_mem" -ResourceGroupName $rgname -ServerNa
me $serverName -Value 262144
Update-AzPostgreSQLConfiguration -Name "max_connections" -ResourceGroupName $rgname -ServerName $ser
verName -Value 100
Update-AzPostgreSQLConfiguration -Name "shared_buffers" -ResourceGroupName $rgname -ServerName $serve
rName -Value 262144
Update-AzPostgreSQLConfiguration -Name "wal_buffers" -ResourceGroupName $rgname -ServerName $serverNa
me -Value 8192
Update-AzPostgreSQLConfiguration -Name "max_wal_size" -ResourceGroupName $rgname -ServerName $server
Name -Value 5120
Update-AzPostgreSQLConfiguration -Name "min_wal_size" -ResourceGroupName $rgname -ServerName $server
Name -Value 2560
```

# Data Migration - Revert Server Parameters

Once the migration has been completed, reset the server parameters to support the target workload. It is possible to run the script to restore on-premises parameters, but it is likely **some of the parameters and corresponding values will not be supported in Azure**.

When attempting to run the settings export snapshot performed at the beginning of the migration, many of the items will cause errors in Azure.

## Configure Server Parameters

The following parameters can be changed on the Azure Database for PostgreSQL target instance. These parameters can be set through the Azure Portal or by using the Azure PowerShell for PostgreSQL cmdlets.

In the absence of a specific workload configuration, as a default, revert the Azure target back to its default settings:

- wal_buffers: 8192.
- max_wal_size: 5120MB.
- min_wal_size: 2560MB.

```
$rgName = "{RESOURCE_GROUP_NAME}";
$serverName = "{SERVER_NAME}";
Update-AzPostgreSQLConfiguration -Name wal_buffers -ResourceGroupName $rgname -ServerName $serverNa
me -Value 8192
Update-AzPostgreSQLConfiguration -Name max_wal_size -ResourceGroupName $rgname -ServerName $serverN
ame -Value 5120
Update-AzPostgreSQLConfiguration -Name min_wal_size -ResourceGroupName $rgname -ServerName $serverNa
me -Value 2560
```

# Data Migration - Application Settings

## Setup

Follow all the steps in the Appendix A: Environment Setup guide to create an environment to support the following steps.

## Update Applications to support SSL

- Switch to the Java Server API (conferencedemo) in Visual Studio code.
- Open the **launch.json** file.
- Update the **DB_CONNECTION_URL** to jdbc:PostgreSQL://servername:5432/reg_app?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC&verifyServerCertificate=true&useSSL=true&requireSSL=true&noAccessToProcedureBodies=true. Note the additional SSL parameters.
- Update **DB_USER_NAME** to **conferenceuser@servername**.
- Start the debug configuration and ensure that the application works locally with the new database.

## Change Connection String for the Java API

- Use the following commands to change the connection string for the App Service Java API.

```
$rgName = "{RESOURCE_GROUP_NAME}";
$app_name = "{SERVER_NAME}";
az webapp config appsettings set -g $rgName -n $app_name --settings DB_CONNECTION_URL={DB_CONNECTION_URL}
```

**Note:** Remember that the Azure Portal can be used to set the connection string.

- Restart the App Service API.

```
az webapp restart -g $rgName -n $app_name
```

You have successfully completed an on-premises to Azure Database for PostgreSQL migration!

# Post Migration Management

## Monitoring and Alerts

Once the migration has successfully completed, the next phase to manage are the new cloud-based data workload resources. Management operations include both the control and data plane activities. Control plane activities manage Azure resources. The data plane is used to interact with the resources, which is **inside** the Azure resource (in this case PostgreSQL).

Azure Database for PostgreSQL provides for the ability to monitor both of these types of operational activities using Azure-based tools such as Azure Monitor, Log Analytics and Azure Sentinel. In addition to the Azure-based tools, security information and event management (SIEM) systems can be configured to consume these logs as well.

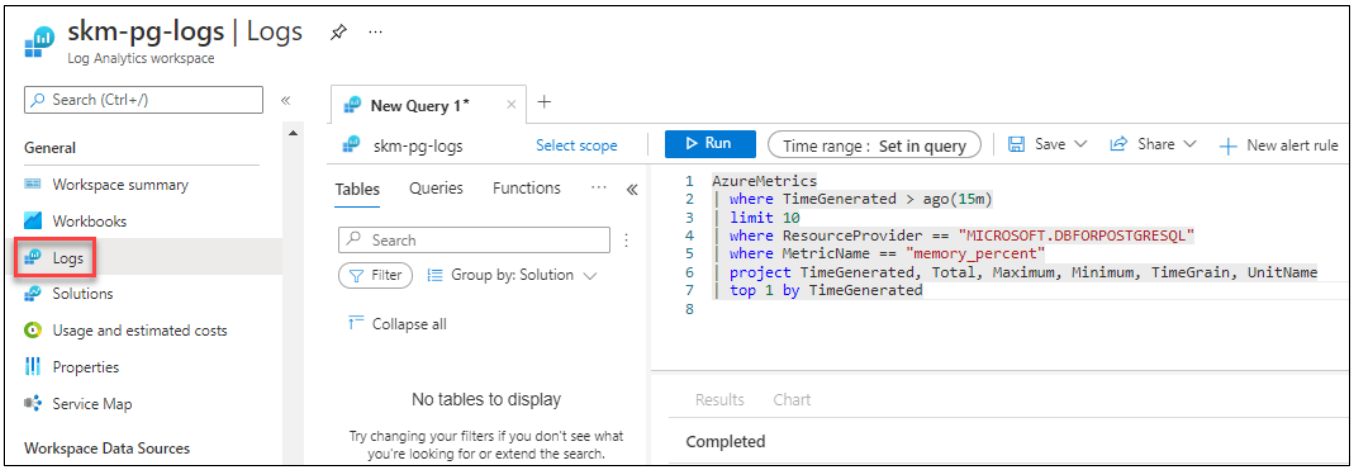
Cloud-based workload monitoring will need to be configured to warn Azure and database administrators of any suspicious activity. If a particular alert event has a well-defined remediation path, alerts can fire automated Azure run books to address the event.

The first step to creating a fully monitored environment is to enable PostgreSQL log data to flow into Azure Monitor. Reference Configure and access audit logs for Azure Database for PostgreSQL in the Azure portal for more information.

Once log data is flowing, use the Kusto Query Language (KQL) query language to query the various log information. Administrators unfamiliar with KQL can find a SQL to KQL cheat sheet here or the Get started with log queries in Azure Monitor page.

For example, to get the memory usage of the Azure Database for PostgreSQL, use a query like the one below.

```
AzureMetrics
| where TimeGenerated > ago(15m)
| limit 10
| where ResourceProvider == "MICROSOFT.DBFORPOSTGRESQL"
| where MetricName == "memory_percent"
| project TimeGenerated, Total, Maximum, Minimum, TimeGrain, UnitName
| top 1 by TimeGenerated
```




KQL query in Log Analytics.

To get the CPU usage:

```
AzureMetrics
| where TimeGenerated > ago(15m)
| limit 10
| where ResourceProvider == "MICROSOFT.DBFORPOSTGRESQL"
| where MetricName == "cpu_percent"
| project TimeGenerated, Total, Maximum, Minimum, TimeGrain, UnitName
| top 1 by TimeGenerated
```

**Note**: for a list of other metrics, reference Monitor and tune Azure Database for PostgreSQL - Single Server.

Once a KQL query has been created, create log alerts based on these queries.

## Server Parameters

As part of the migration, it is likely the on-premises server parameters were modified to support a fast egress. Also, modifications were made to the Azure Database for PostgreSQL parameters to support a fast ingress. The Azure server parameters should be set back to their original on-premises workload optimized values after the migration.

However, be sure to review and make server parameters changes that are appropriate for the workload and the environment. Some on-premises environment values may not be optimal for a cloud-based environment. Verify on-premises parameters can be set in the Azure environment.

## PowerShell Module

The Azure Portal and Windows PowerShell can be used for managing the Azure Database for PostgreSQL. To get started with PowerShell, install the Azure PowerShell **cmdlets** for PostgreSQL with the following PowerShell command:

```
Install-Module -Name Az.PostgreSql
```

After the modules are installed, reference tutorials and documentation like the following to learn ways to take advantage of scripting various management activities:

- Tutorial: Design an Azure Database for PostgreSQL using PowerShell
- Restore an Azure Database for PostgreSQL server using PowerShell
- Configure server parameters in Azure Database for PostgreSQL using PowerShell
- How to create and manage read replicas in Azure Database for PostgreSQL using PowerShell
- Restart Azure Database for PostgreSQL server using PowerShell

## Azure Database for PostgreSQL Upgrade Process

Since Azure Database for PostgreSQL is a PaaS offering, administrators are not responsible for applying operating system and PostgreSQL updates. However, the upgrade process can be random. PostgreSQL server workloads will be stopped during the upgrade deployment. Plan for these downtimes by rerouting the workloads to a read replica in the event the instance goes into maintenance mode.

**Note:** This style of failover architecture may require changes to the application's data layer to support this type of failover scenario. If the read replica is maintained as a read replica and is not promoted, the application will only be able to read data. Database write operations will fail.

The Planned maintenance notification feature will inform resource owners up to 72 hours in advance of installation of an update or critical security patch. Database administrators may need to notify application users of planned and unplanned maintenance.

**Warning:** Azure Database for PostgreSQL maintenance notifications are incredibly important. The database maintenance can take the database and connected applications down for a random period of time.

## WWI Use Case

WWI decided to utilize the Azure Activity logs and enable PostgreSQL logging to flow to a Log Analytics workspace. This workspace is configured to be a part of Azure Sentinel such that any Threat Analytics events would be surfaced, and incidents created.

## Management Checklist

- Create resource alerts for common things like CPU and Memory.
- Ensure the server parameters are configured for the target data workload after migration.
- Script common administrative tasks.
- Set up notifications for maintenance events such as upgrades and patches. Notify users.

# Optimization

## Monitoring Hardware and Query Performance

In addition to the audit and activity logs, server performance can also be monitored with Azure Metrics. Azure metrics are provided in a one-minute frequency, and alerts can be configured from them. For more information, reference Monitoring in Azure Database for PostgreSQL for specifics on what kind of metrics that can be monitored.

As previously mentioned, monitoring metrics such as the cpu_percent or memory_percent can be important when deciding to upgrade the database tier. Consistently high values could indicate a tier upgrade is necessary.

Additionally, if CPU and memory do not seem to be the issue, administrators can explore database-based options such as indexing and query modifications for poor performing queries.

To find poor performing queries, run the following:

```
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.DBFORPOSTGRESQL"
| where Category == 'PostgreSQLSlowLogs'
| project TimeGenerated, LogicalServerName_s, event_class_s, start_time_t , query_time_d, sql_text_s
| top 5 by query_time_d desc
```

### pg_stat_statements Table

Additionally, use the following PostgreSQL command to query the pg_stat_statements table and show the top 5 duration queries during a benchmarking run:

```
SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit/
nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
FROM pg_stat_statements
ORDER BY total_time
DESC LIMIT 5
```

## Typical Workloads and Performance Issues

There tends to be two common usage patterns with any database system, PostgreSQL included. These are (but are not limited to):

- An application server exposing a web endpoint on an application server, which connects to the database.
- A client-server architecture where the client directly connects to the database.

In consideration of the above patterns, performance issues can crop up in any of the following areas:

- **Resource Contention (Client)** - The machine/server serving as the client can constrain resources. The constraints can be found in in the Azure VM task manager, the Azure portal, or Azure CLI.
- **Resource Contention (Application Server)** - The machine/server acting as the application server can constrain resources. The Azure VM task manager, the Azure portal, or CLI can identify the constrained resources. If the application server is an Azure service or virtual machine, then Azure Metrics can help with determining the resource contention.
- **Resource Contention (Database Server)** - The database service could be experiencing performance bottlenecks related to CPU, memory, and storage, which can be determined from the Azure Metrics for the database service instance.
- **Network latency** - A quick SQL network check should be done before starting any performance benchmarking. To determine the network latency between the client and database, execute a simple SELECT 1 query. In most Azure regions, watch for **less than two milliseconds** of latency on SELECT 1 timing when using a remote client hosted on Azure in the same region as the Azure Database for PostgreSQL server.

## Query Performance Insight (QPI)

In addition to the basic server monitoring aspects, Azure provides tools to monitor application query performance. Improving query performance can lead to significant increases in the query throughput. Use the Query Performance Insight tool to analyze the longest running queries. Consider caching query output for a deterministic set period, or modify the queries to increase their performance.

Set the slow_query_log server parameter to show slow queries in the PostgreSQL log files (default is OFF). The long_query_time server parameter can alert users for long query times (default is 10 sec).

## Query Store (QS)

The Query Store feature in Azure Database for PostgreSQL provides a way to track query performance over time. Query Store simplifies performance troubleshooting by helping quickly find the longest running and most resource-intensive queries.

The Query Store is not enabled by default and must be turned on via the query_capture_mode server parameter. Once enabled, query the query_store.qs_view table in the azure_sys database for runtime execution statistics.

**Note**: In order to use both QS and QPI, enable data collection by setting pg_qs.query_capture_mode and pgms_wait_sampling.query_capture_mode server parameters to ALL.

### Regressed Queries

Regressed queries can be a real problem. The Query Store enables the monitoring of regressed queries. By setting pg_qs.query_capture_mode to ALL, a history of the query performance over time will become available. This data can be leveraged for simple or more complex performance comparisons. A regressed query list requires setting a comparison period. During the comparison period, query runtime statistics are baselined for future comparisons.

There are a handful of factors to think about when selecting the comparison period:

- **Seasonality**: Does target query occur periodically rather than continuously?
- **History**: Is there enough historical data?
- **Threshold**: Is it acceptable to use a flat percentage change threshold? Or is it required to use a more complex method to prove the statistical significance of the regression?

Create a function to get query performance changes:

```
create or replace function get_ordered_query_performance_changes(
baseline_interval_start int,
baseline_interval_type text,
current_interval_start int,
current_interval_type text)
returns table (
    query_id bigint,
    baseline_value numeric,
    current_value numeric,
    percent_change numeric
) as $$
with data_set as (
select query_id
, round(avg( case when start_time >= current_timestamp - ($1 || $2)::interval and start_time < current_timestamp
- ($3 || $4)::interval then mean_time else 0 end )::numeric,2) as baseline_value
, round(avg( case when start_time >= current_timestamp - ($3 || $4)::interval then mean_time else 0 end )::numer
ic,2) as current_value
from query_store.qs_view where query_id != 0 and user_id != 10 group by query_id ) ,
query_regression_data as (
select *
, round(( case when baseline_value = 0 then 0 else (100*(current_value - baseline_value) / baseline_value) end )::n
umeric,2) as percent_change
```

```
from data_set )
select * from query_regression_data order by percent_change desc;
$$
language 'sql';
```

This query can be run by executing the following to compare a baseline over the last seven days up to two hours ago:

```
select * from get_ordered_query_performance_changes (7, 'days', 2, 'hours');
```

The top changes returned from this query are all good candidates to review unless it is an expected delta from the baseline period. Once queries have been identified, the next step is to look further into query store data and see how the baseline statistics compare to the current period and collect additional clues.

### The Oversized-Attribute Storage Technique (TOAST)

PostgreSQL uses a fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages. Due to this restriction, it is not possible to store very large field values directly. To overcome this limitation, large field values are compressed and/or broken up into multiple physical rows. This happens transparently with only small impact on most of the backend code. In addition to overcoming the page size limitation, it is also used to improve handling of large data values in-memory. This strategy is known as TOAST.

Only certain data types support TOAST — there is no need to impose the overhead on data types that cannot produce large field values. To support TOAST, a data type must have a variable-length (varlena) representation. Normally, the first four-byte word of any stored value contains the total length of the value in bytes (including itself).

Four different strategies are used to store columns on disk that use TOAST. They represent various combinations, between compression and out-of-line storage. The strategy can be set at the level of data type and at the column level.

- **Plain** prevents either compression or out-of-line storage. It disables the use of single-byte headers for varlena types. Plain is the only possible strategy for columns of data types that cannot use TOAST.
- **Extended** allows both compression and out-of-line storage. Extended is the default for most data types that can use TOAST. Compression is attempted first. Out-of-line storage is attempted if the row is still too large.
- **External** allows out-of-line storage but not compression. Use of External makes substring operations on wide text and bytea columns faster. This speed comes with the penalty of increased storage space. These operations are optimized to fetch only the required parts of the out-of-line value when it is not compressed.
- **Main** allows compression but not out-of-line storage. Out-of-line storage is still performed for such columns, but only as a last resort. It occurs when there's no other way to make the row small enough to fit on a page.

If queries access data types that can use TOAST, consider using the Main strategy instead of the default Extended option to reduce query times. Main does not rule out out-of-line storage. If queries do not access data types that can use TOAST, it might be beneficial to keep the Extended option. A larger portion of the rows of the main table fit in the shared buffer cache, which helps performance.

With a workload that uses a schema with wide tables and high character counts, consider using PostgreSQL TOAST tables. An example customer table had greater than 350 columns with several columns that spanned 255 characters. After it was converted to the TOAST table Main strategy, their benchmark query time reduced from 4203 seconds to 467 seconds. That is an 89 percent improvement.

## Performance Recommendations

The Performance Recommendations feature analyzes workloads across the server to identify indexes with the potential to improve performance. The tool requires the Query Store feature to be enabled. Once enabled, manually review and implement any suggestions.

## Azure Advisor

Azure Advisor will provide recommendations around Azure Database for PostgreSQL Performance, Reliability and Cost. For example, Azure Advisor could present a recommendation to modify a server parameter based on the workloads and instance pricing tier and available scaling settings.

## Upgrading the Tier

The Azure Portal can be used to scale from General Purpose and Memory Optimized. If a Basic tier is chosen, there will be no option to upgrade the tier to General Purpose or Memory Optimized later. However, it is possible to utilize other techniques to perform a migration/upgrade to a new Azure Database for PostgreSQL instance.

For an example of a script that will migrate from Basic to another server tier, reference Upgrade from Basic to General Purpose or Memory Optimized tiers in Azure Database for PostgreSQL.

## Scale the Server

Within the tier, it is possible to scale cores and memory to the minimum and maximum limits allowed in that tier. If monitoring shows a continual maxing out of CPU or memory, follow the steps to Manage an Azure Database for PostgreSQL server using Azure Portal.

## Moving Regions

Moving a database to a different Azure region depends on the approach and architecture. Depending on the approach, it could cause system downtime.

The recommended process is the same as utilizing read replicas for maintenance failover. However, compared to the planned maintenance method mentioned above, the speed to failover is much faster when a failover layer has

been implemented in the application. The application should only be down for a few moments during the read replica failover process. More details are covered in the Business Continuity and Disaster Recovery (BCDR) section.

## Quick Tips

Use the following to make quick performance changes:

- **CPU Usage**: If CPU usage for an Azure Database for PostgreSQL server is saturated at 100%, then select the next higher level of Compute Units to get more CPU.
- **IOPS**: The default storage size of 125GB is limited to 375 IOPs. If the application requires higher IOPs, then it is recommended that a higher storage size be selected to get more IOPs.
- **Regions**: It is recommended having the application server/client machine in the same region in Azure to reduce latency between the client/application server and the database.
- **Accelerated Networking**: Accelerated networking enables single root I/O virtualization (SR-IOV) to a VM, greatly improving its networking performance. This high-performance path bypasses the host from the data path, reducing latency, jitter, and CPU utilization for use with the most demanding network workloads on supported VM types.

## WWI Use Case

WWI business and application users expressed excitement regarding the ability to scale the database on-demand. They were also interested in using the Query Performance Insight to determine if long running queries' performance needed to be addressed.

They opted to utilize a read replica server for any potential failover or read-only scenarios.

The migration team, working with the Azure engineers, set up KQL queries to monitor for any potential issues with the PostgreSQL server performance. The KQY queries were set up with alerts to email event issues to the database and conference team.

They elected to monitor any potential issues for now and implement Azure Automation run books later, if needed, to improve operational efficiency.

## Optimization Checklist

- Enable Query Store
- Monitor for slow queries.
- Periodically review the Performance and Azure Advisor recommendations.
- Utilize monitoring to drive tier upgrades and scale decisions.

- Consider moving regions if the users' or application's needs change.

# Business Continuity and Disaster Recovery (BCDR)

## Backup and Restore

As with any mission critical system, having a backup and restore and a disaster recovery (BCDR) strategy is an important part of the overall system design. If an unforeseen system failure event occurs, it is important to have the ability to restore the data to a point-in-time (Recovery Point Objective) and in a reasonable amount of time (Recovery Time Objective). How much time and data can you afford to lose?

### Backup

**Azure Database for PostgreSQL supports automatic backups for 7 days by default**. It may be appropriate to modify this to the current maximum of 35 days. It is important to be aware that if the value is changed to 35 days, there will be charges for any extra backup storage over 1x the storage allocated.

There are several limitations to the database backup features as described in each of the backup articles for each service type. It is important to understand them when deciding what additional strategies that should be implemented:

- Backup and restore in Azure Database for PostgreSQL - Single Server
- Backup and restore in Azure Database for PostgreSQL - Flexible Server
- Backup and restore in Azure Database for PostgreSQL - Hyperscale Citus

Commonalities of the backup architectures include:

- Up to 35 days of backup protection
- No direct access to the backups (no exports).

Some items to be aware of include:

- Tiers that allow up to 4TB will retain two full backups, all differential backups, and transaction logs since the last full backup every 7 days.
- Tiers that allow up to 16TB will retain the full backup, all differential backups, and transaction logs in the last 8 days.

**Note:** Some regions do not yet support storage up to 16TB.

### Restore

Redundancy (local or geo) must be configured during server creation. However, a geo-restore can be performed and allows the modification of these options during the restore process. Performing a restore operation will temporarily stop connectivity and any applications will be down during the restore process.

During a database restore, any supporting items outside of the database will also need to be restored. See Perform post-restore tasks for more information.

## Replicas

### Read Replicas (Single Server)

Read replicas can be used to increase the PostgreSQL read throughput, improve performance for regional users, and implement disaster recovery. When creating one or more read replicas, be aware that additional charges will apply for the same compute and storage as the primary server.

### Standby Replicas (Flexible Server)

When deployed in a zone redundant configuration, flexible server automatically provisions and manages a standby replica in a different availability zone. Using PostgreSQL streaming replication, the data is replicated to the standby replica server in synchronous mode.

**Note**: Availability zones are within the same region. If an entire region were to go down, a secondary BCDR strategy will be needed to recover quickly from the regional failure. At this time (4/2021), cross-region read replica and geo-restore of backup features are not yet supported in preview.

### Read Replicas (Hyperscale Citus)

The read replica feature allows for replicated data from a Hyperscale Citus server group to a read-only server group. Replicas are updated asynchronously with PostgreSQL physical replication technology. It is possible to replicate from the primary server to an unlimited number of replicas.

**Note**: Read replicas are in preview as of 4/2021.

## Deleted Servers

If an administrator or bad actor deletes the server in the Azure Portal or via automated methods, all backups and read replicas will also be deleted. It is important that resource locks are created on the Azure Database for PostgreSQL resource group to add an extra layer of deletion prevention to the instances.

## Regional Failure

Although rare, if a regional failure occurs, geo-redundant backups or a read replica can be used to get the data workloads running again. It is best to have both geo-replication and a read replica available for the best protection against unexpected regional failures.

**Note**: Changing the database server region also means the endpoint will change and application configurations will need to be updated accordingly.

### Load Balancers

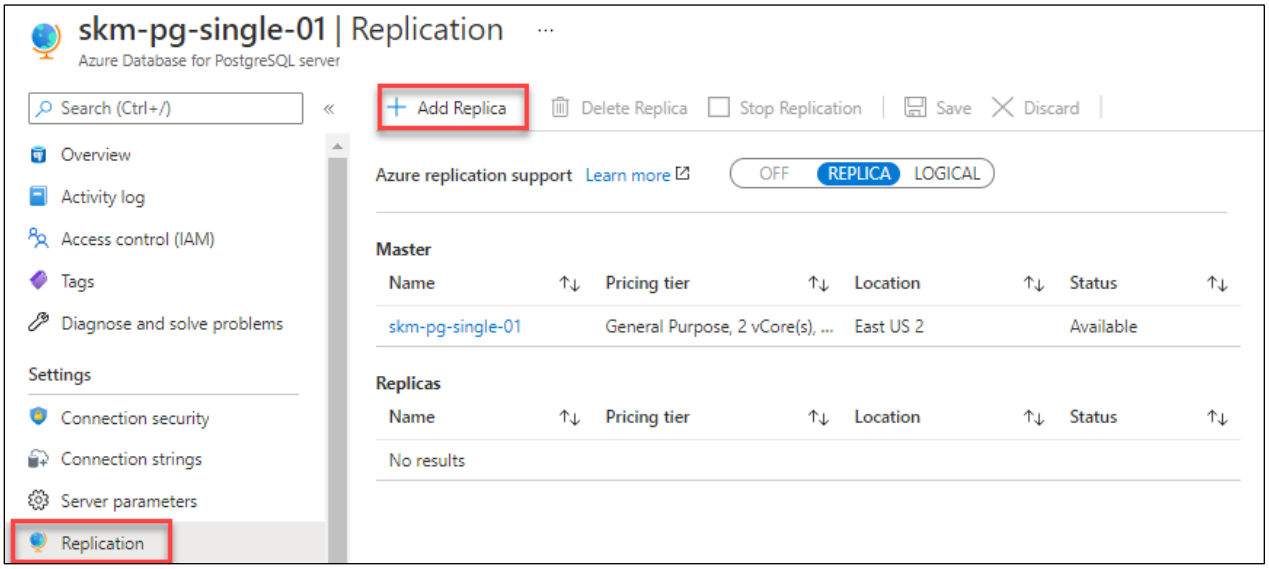
If the application is made up of many different instances around the world, it may not be feasible to update all of the clients. Utilize an Azure Load Balancer or Application Gateway to implement a seamless failover functionality. Although helpful and timesaving, these tools are not required for regional failover capability.

## WWI Use Case

WWI wanted to test the failover capabilities of read replicas, so they performed the steps outlined below.

### Creating a read replica (Single Server)

- Open the Azure Portal.
- Browse to the Azure Database for PostgreSQL instance.
- Under **Settings**, select **Replication**.
- Select **Add Replica**.




Configure replication for Azure Database for PostgreSQL single server.

- Type a server name.
- Select the region.
- Select **OK**, wait for the instance to deploy. Depending on the size of the main instance, it could take some time to replicate.

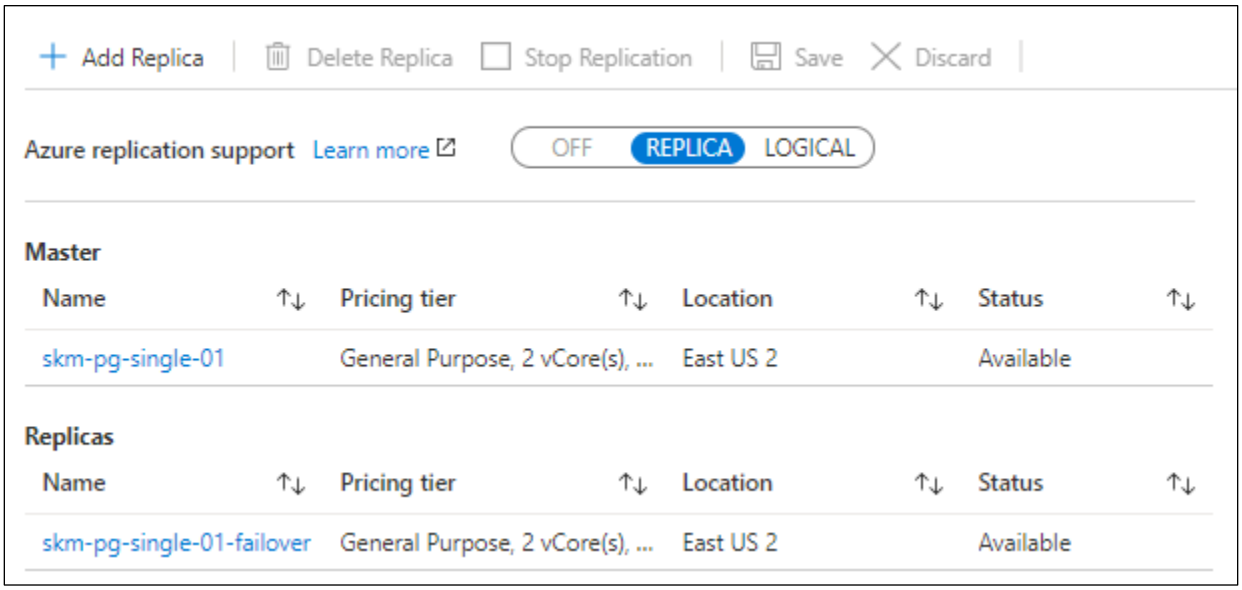
**Note:** Each replica will incur additional charges equal to the main instance.

### Failover to read replica (Single Server)

Once a read replica has been created and has completed the replication process, it can be used for failed over. Replication will stop during a failover and make the read replica its own main instance.

Failover Steps:

- Open the Azure Portal.
- Browse to the Azure Database for PostgreSQL instance.
- Under **Settings**, select **Replication**.
- Select one of the read replicas.
- Select **Stop Replication**. This will break the read replica.
- Modify all applications' connection strings to point to the new main instance.




Read replica in Azure portal.

## BCDR Checklist

- Modify backup frequency to meet requirements.
- Setup read replicas for read intensive workloads and regional failover.
- Create resource locks on resource groups.
- Implement a load balancing strategy for applications for quick failover.

# Security

Moving to a cloud-based service doesn't mean the entire internet will have access to it at all times. Azure provides best in class security that ensures data workloads are continually protected from bad actors and rogue programs.

## Authentication

Azure Database for PostgreSQL supports the basic authentication mechanisms for PostgreSQL user connectivity, but also supports integration with Azure Active Directory. This security integration works by issuing tokens that act like passwords during the PostgreSQL login process. Configuring Active Directory integration is incredibly simple to implement and supports not only users, but AAD groups as well.

This tight integration allows administrators and applications to take advantage of the enhanced security features of Azure Identity Protection to further surface any identity issues.

**Note:** Be sure to test the application with Azure AD Authentication. See Use Azure Active Directory for authentication with PostgreSQL for more information.

## Threat Protection

If user or application credentials are compromised, logs are not likely to reflect any failed login attempts. Compromised credentials can allow bad actors to access and download the data. Azure Threat Protection can watch for anomalies in logins (such as unusual locations, rare users or brute force attacks) and other suspicious activities. Administrators can be notified in the event something does not look right.

## Audit Logging

PostgreSQL has a robust built-in logging feature. By default, this log feature is disabled in Azure Database for PostgreSQL. Server level logging can be enabled or modified by changing various server parameters. Once enabled, logs can be accessed through Azure Monitor and Log Analytics by turning on diagnostic logging.

To query for log related events, run the following KQL query:

```
AzureDiagnostics
| where LogicalServerName_s == "myservername"
| where Category == "PostgreSQLLogs"
| where TimeGenerated > ago(1d)
```

In addition to the basic logging feature, gain access to more in-depth audit logging information that is provided with the pgaudit extension.

```
AzureDiagnostics
| where LogicalServerName_s == "myservername"
```

```
| where TimeGenerated > ago(1d)
| where Message contains "AUDIT:"
```

## Encryption

Data in the PostgreSQL instance is encrypted at rest by default. Any automated backups are also encrypted to prevent potential leakage of data to unauthorized parties. This encryption is typically performed with a key that is created when the instance is created. In addition to this default encryption key, administrators have the option to bring their own key (BYOK).

When using a customer-managed key strategy, it is vital to understand responsibilities around key lifecycle management. Customer keys are stored in an Azure Key Vault and then accessed via policies. It is vital to follow all recommendations for key management as the loss of the encryption key equates to the loss of data access.

In addition to customer-managed keys, use service-level keys to add double encryption. Implementing this feature will provide highly encrypted data at rest, but it does come with encryption performance penalties. Testing should be performed.

Data can be encrypted during transit using SSL/TLS. As previously discussed, it may be necessary to modify your applications to support this change and also configure the appropriate TLS validation settings.

## Firewall

Once users are set up and the data is encrypted at rest, the migration team should review the network data flows. Azure Database for PostgreSQL provides several mechanisms to secure the networking layers by limiting access to only authorized users, applications, and devices.

The first line of defense for protecting the PostgreSQL instance is to implement firewall rules. IP addresses can be limited to only valid locations when accessing the instance via internal or external IPs. If the PostgreSQL instance is destined to only serve internal applications, then restrict public access.

When moving an application to Azure along with the PostgreSQL workload, it is likely there will be multiple virtual networks setup in a hub and spoke pattern that will require Virtual Network Peering to be configured.

## Private Link

To limit access to the Azure Database for PostgreSQL to internal Azure resources, enable Private Link. Private Link will ensure that the PostgreSQL instance is assigned a private IP address, rather than a public IP address.

**Note**: Not all Azure database for PostgreSQL services support private link (4/2021).

**Note**: There are many other basic Azure Networking considerations that must be taken into account that are not the focus of this guide.

## Security baseline

Review a set of potential security baseline tasks that can be implemented across all Azure resources. Not all the items described on the reference link will apply to the specific data workloads or Azure resources.

## Security Checklist

- Use Azure AD authentication where possible.
- Enable Advanced Thread Protection.
- Enable all auditing features.
- Consider a Bring-Your-Own-Key (BYOK) strategy.
- Implement firewall rules.
- Utilize private endpoints for workloads that do not travel over the Internet.

# Summary

This document has covered several topics related to migrating an application from on-premises PostgreSQL to Azure Database for PostgreSQL. We covered how to begin and assess the project all the way to application cut over.

The migration team will need to review the topics carefully as the choices made can have project timeline effects. The total cost of ownership is very attractive given the many enterprise-ready features provided.

The migration project approach is very important. The team will need to assess the application and database complexity to determine the amount of conversion time. Conversion tools will help make the transition easier, but there will always be an element of manual review and updates required. Scripting out pre-migration tasks and post-migration testing is important.

Application architecture and design can provide strong indicators as to the level of effort required. For example, applications utilizing ORM frameworks can be great candidates, especially if the business logic is contained in the application instead of database objects.

In the end, several tools exist in the marketplace ranging from free to commercial. This document covered the steps required if the team plans a database migration using one of the more popular open-source tool options. Whichever path that is chosen, Microsoft and the PostgreSQL community have the tools and expertise to make the database migration successful.

## Questions and Feedback

For any questions or suggestions about working with Azure Database for PostgreSQL, send an email to the Azure Database for PostgreSQL Team (AskAzureDBforPostgreSQL@service.microsoft.com). Please note that this address is for general questions rather than support tickets.

In addition, consider these points of contact as appropriate:

- To contact Azure Support or fix an issue with your account, file a ticket from the Azure portal.
- To provide feedback or to request new features, create an entry via UserVoice.

## Find a partner to assist in migrating

This guide can be overwhelming, but don't fret! There are many experts in the community with a proven migration track record. Search for a Microsoft Partner or Microsoft MVP to help with finding the most appropriate migration strategy. You are not alone!

Browse the technical forums and social groups for more detailed real-world information:

- Microsoft Community Forum

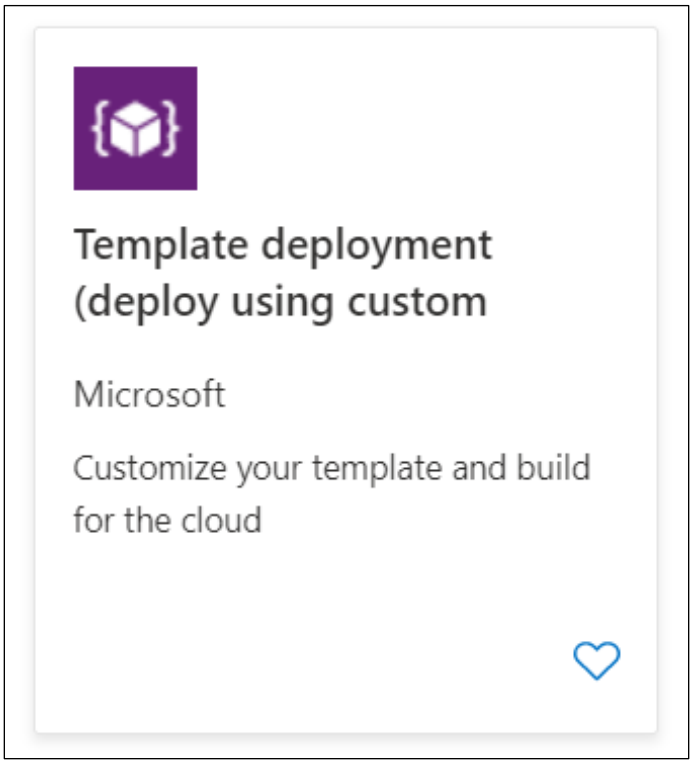
- StackOverflow for Azure PostgreSQL
- Azure Facebook Group
- LinkedIn Azure Group
- LinkedIn Azure Developers Group

# Appendix A: Environment Setup

The following steps will configure an environment to perform the guide's migration steps.

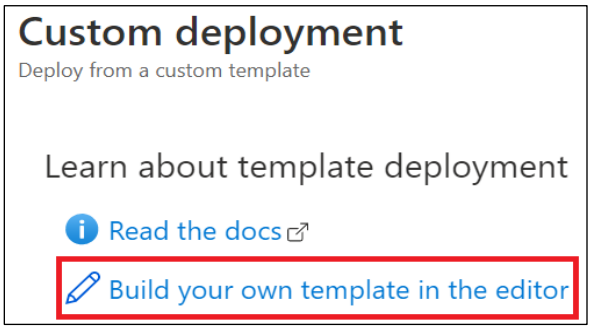
## Deploy the ARM template

- Open the Azure Portal
- Create a new resource group
- Select **+Add**, type **template**, select the **Template Deployment...**




Selecting the template deployment option in the Azure portal.

- Select **Create**
- Select **Build your own template in the editor**




Adding the custom ARM template to the template deployment page in the Azure portal.

- Choose between the Configure Network Security (Secure path) or the Deploy the ARM template ARM template. The difference between the two options is the secured option's resources are hidden behind an App Gateway with private endpoints, whereas with the other, resources are directly exposed to the internet.

**Note**: The secure template runs at ~$1700 per month. The non-secure template runs at ~$700 per month.

- Copy the json into the window
- Select **Save**

The added template is visible in the ARM template editor window.

- Fill in the parameters
  - Be sure to record your prefix and password, as they are needed later
- Select **Review + create**
- Select the **I agree…** checkbox
- Select **Create**. After about 20 minutes, the landing zone will be deployed

## Open the Azure VM Ports

- Browse to the Azure Portal.
- Select the **PREFIX-vm-pgdb01** virtual machine resource.
- Under **Settings**, select **Networking**
- In the **Inbound port rules** area, select **Add inbound port rule**
- For the **Destination port ranges**, type **5432**
- For the name, type **Port_5432**
- Select **Add**

## Allow Azure PostgreSQL Access

- Browse to the Azure Portal.
- Select the **PREFIX-pg-single-01** instance.
- Under **Settings**, select **Connection security**
- Toggle the **Allow access to Azure services** to **On**
- Select **Save**
- Browse to your resource group
- Select the **PREFIX-pg-flex-01** instance.
- Under **Settings**, select **Networking**
- Toggle the **Allow public access from any Azure service within Azure to this server** to **On**
- Select **Save**

## Connect to the Azure VM

- Login to the deployed database VM
  - Browse to the Azure Portal.
  - Select the **PREFIX-vm-pgdb01** virtual machine resource.
  - Select **Connect->RDP**
  - Select **Open** in the RDP dialog

  - Login using s2admin and Seattle123Seattle123

## Install Chrome

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Open a browser window, browse to the Chrome download
- Click **Download Chrome**
- Follow all the prompts

## Install ActivePerl 5.26 (PostgreSQL 10.16)

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Open a browser window, browse to the ActivePerl installer
- Download the ActivePerl 5.26 installer
- Start the installer and click through the installer dialogs

## Install Python 3.7.4 (PostgreSQL 10.16)

Depending on the version of PostgreSQL, in this case 10.16, you install will target different python versions. If you do not download the correctly linked Python version, you will not be able to enable the Python language later.

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Browse here to locate the installer.
- Scroll to the bottom of the page, select the **Windows x86-64 executable installer**

**Note:** You will need to install Python for all users. This will require you to create a customized installation. We recommend installing Python to a directory like C:\Python37.

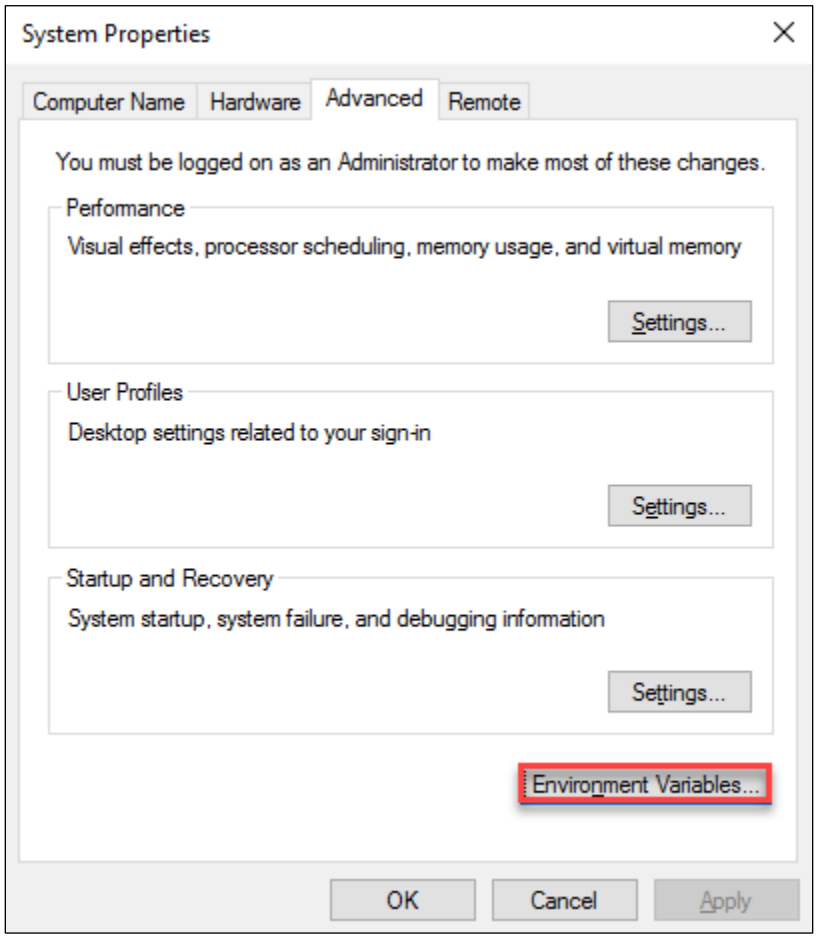
- Check the **Add Python 3.7 to PATH** checkbox
- Select **Customize Installation**
- Select **Next**
- Select **Install for all users**
- Select **Install**
- Select **Close**

## Setup Java and Maven

- Download and Install Java Development Kit 11.0
- Select the **Windows x64 Installer**
- Select **Run**
- Select **Next**
- Select **Next**
- Select **Close**

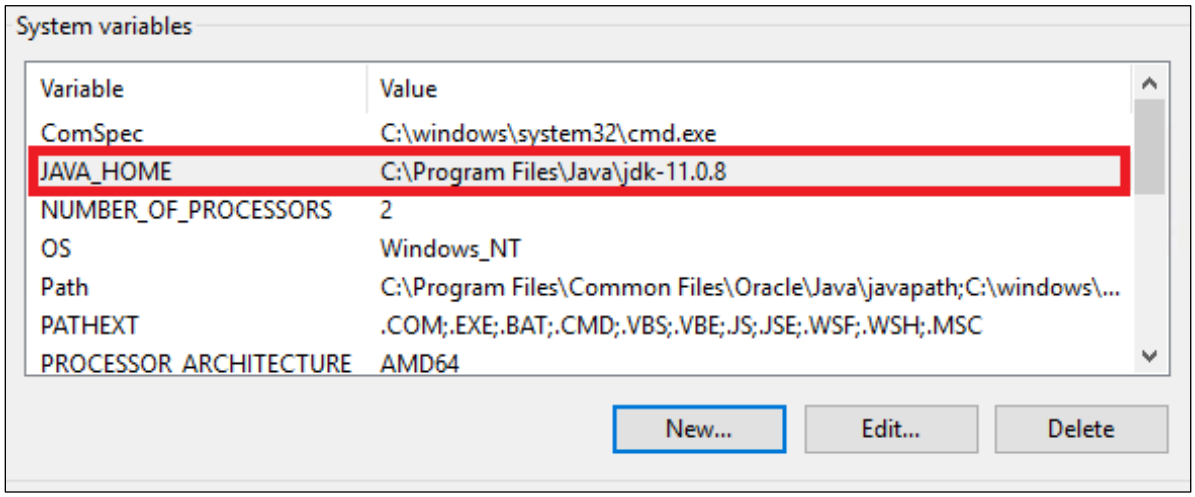
**Note:** Be sure to check what the highest SDK/JRE that is supported in Azure App Service before downloading the latest.

- Set the JAVA_HOME environment variable to the **C:\Program Files\Java\jdk-{version}** folder
  - Open Windows Explorer
  - Right-click **This PC**, select **Properties** and then **Advanced system settings**
  - Select **Environment Variables**




Opening the Environment Variable configuration window in Windows.

- Select **New** under **System variables**
- Type **JAVA_HOME**
- Copy the path shown below, then select **OK.** The image below shows the correct configuration for the **JAVA_HOME** environment variable.




Creating a new JAVA_HOME system environment variable.

- Leave the Environment Variables dialog open
- Download and install the Java Runtime
- Download and configure Maven
  - Download the **binary zip archive**
  - From the download location, right-click the zip archive and select **Extract All...**
  - Set the destination to **C:\Program Files**. Then, select **Extract**
  - Set the M2_HOME environment variable to the **C:\Program Files\apache-maven-{version}** folder
  - Add the **C:\Program Files\apache-maven-{version}\bin** path to the PATH environment variable

## Install PostgreSQL 10.16

- In the Virtual Machine, download the following PostgreSQL versions
  - PostgreSQL 10.16
  - PostgreSQL 11.0
- Install PostgreSQL 10.16
  - Start the PostgreSQL 10.16 installer you just downloaded
  - Select **Run**

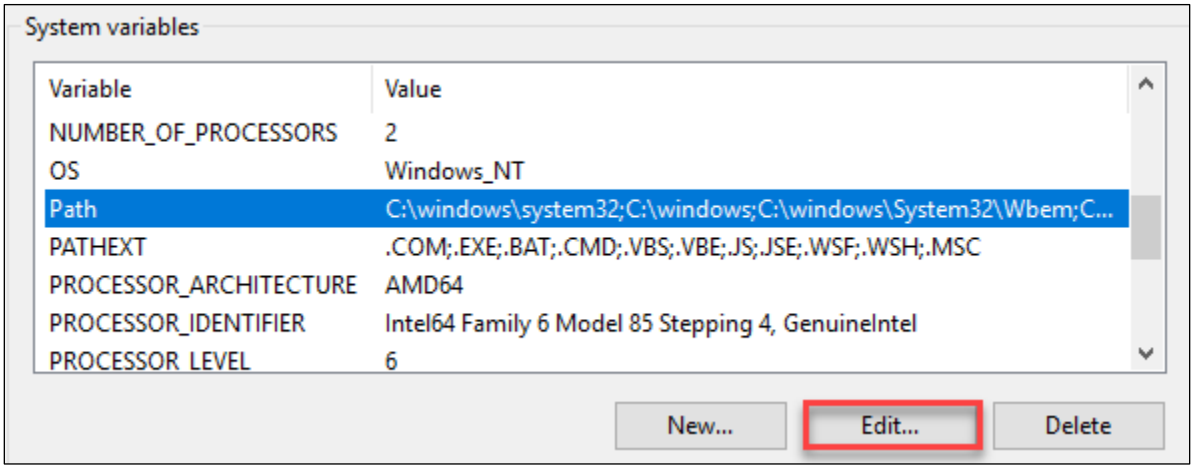
  - On the Welcome dialog, select **Next**
  - On the installation directory dialog, select **Next**
  - On the select components dialog, select **Next**
  - On the data directory dialog, select **Next**
  - For the password, type Seattle123, then select **Next**
  - For the port, select **Next**

  **Note:** The default port is 5432. If you have changed this port, you will need to ensure that you open the necessary paths in the firewall and gateways from Azure to your on-premises\cloud environment.

  - Select your locale, select **Next**
  - On the summary dialog, select **Next**
  - On the ready dialog, select **Next**, PostgreSQL will start the installation process
  - Select **Finish**

- Run StackBuilder
  - Select the local instance, then select **Next**
  - Select the following applications, then select **Next**
    - EDB Language Pack v1.0-5 (expand **Add-ons, tools and utilities**)
    - pgBouncer v1.x (expand **Add-ons, tools and utilities**)
    - pgJDBC (64bit) v13x (expand **Database Drivers**)
    - psqlODBC (64bit) v13x (expand **Database Drivers**)
    - Migration Toolkit v54.x (expand **Registration-required and trial products > EnterpriseDB Tools**)
    - Replication Server v6.x (also an EnterpriseDB tool)
  - For the download directory dialog, type **C:\Users\s2admin\Downloads**, then select **Next**. The applications will download.
  - Install all the StackBuilder applications by clicking through the dialogs and accepting all the defaults
- Add the **C:\Program Files\PostgreSQL\10\bin** path to the PATH environment variable

- Switch to the Environment Variables window
- Under **System variables**, choose **Path**. Then, select **Edit...**




Editing the PATH system environment variable in Windows to add PostgreSQL utilities.

- In the **Edit environment variable** dialog, select **New** and then **Browse...** Browse to C:\Program Files\PostgreSQL\10\bin.
- Select **OK**.

## Configure Language Packs

The default PostgreSQL installer should also install the programming language packs. These language packs include python, tcl, and perl. By default, they will be installed in c:\ebd.

**Note**: You can find out what version of Python has been targeted with the PostgreSQL build by running the python.exe executable in the language pack folder.

## pgAdmin

- You may find it easier to download the latest pgAdmin tool rather than relying on the PostgreSQL installer. You can find the latest versions here.

## Azure Data Studio

- Download the Azure Data Studio tool
- Install the PostgreSQL Extension
  - Select the extensions icon from the sidebar in Azure Data Studio.

  - Type 'postgresql' into the search bar. Select the PostgreSQL extension.
  - Select **Install**. Once installed, select Reload to activate the extension in Azure Data Studio.

## Download artifacts

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Download and Install Git
  - Download and run the 64-bit installer
  - Click **Next** through all prompts
- Open a Windows PowerShell window (just by entering "PowerShell" into the Start menu) and run the following commands

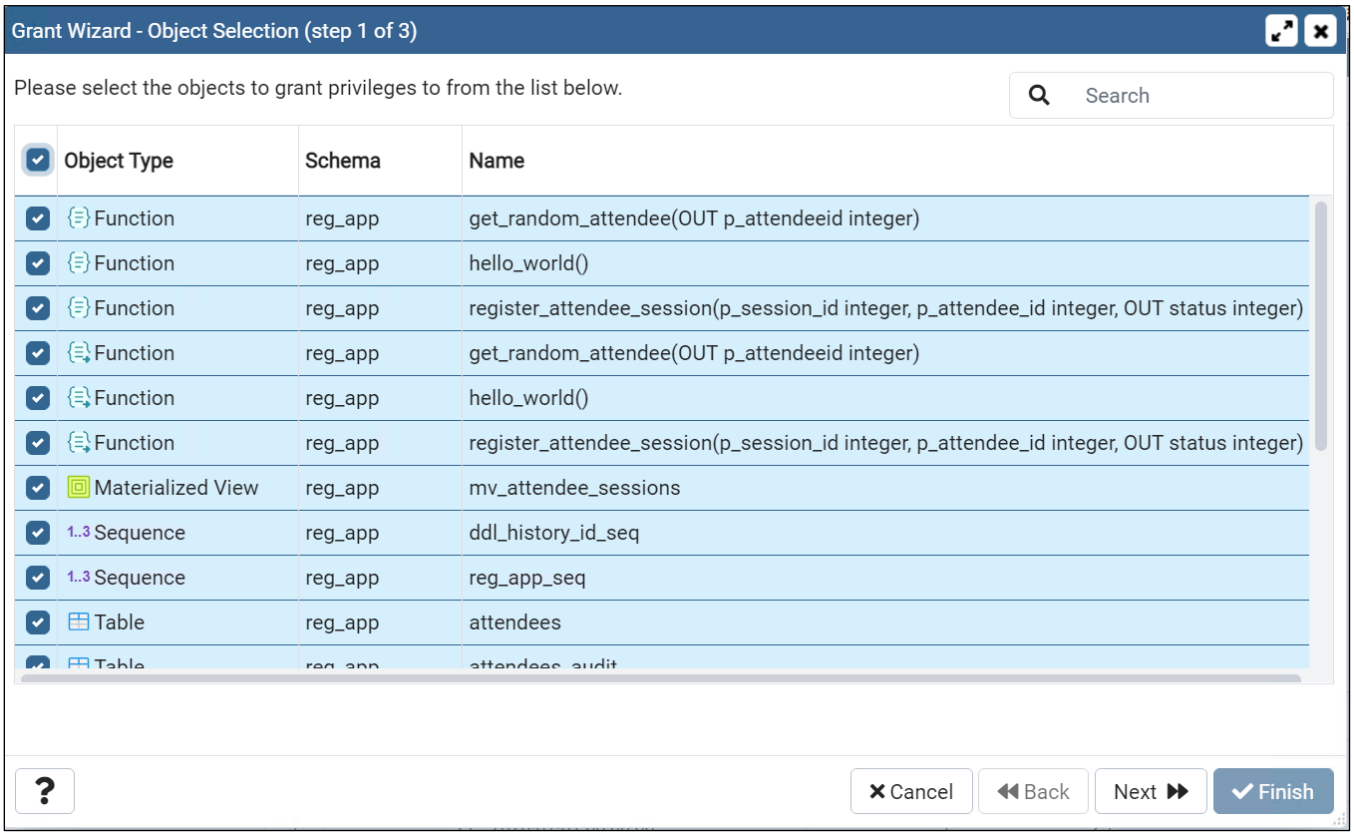
```
mkdir c:\PostgreSQLguide
cd c:\PostgreSQLguide
git config --global user.name "FIRST_NAME LAST_NAME"
git config --global user.email "MY_NAME@example.com"
git clone https://github.com/solliancenet/onprem-postgre-to-azurepostgre-migration-guide
```

## Deploy the Database

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

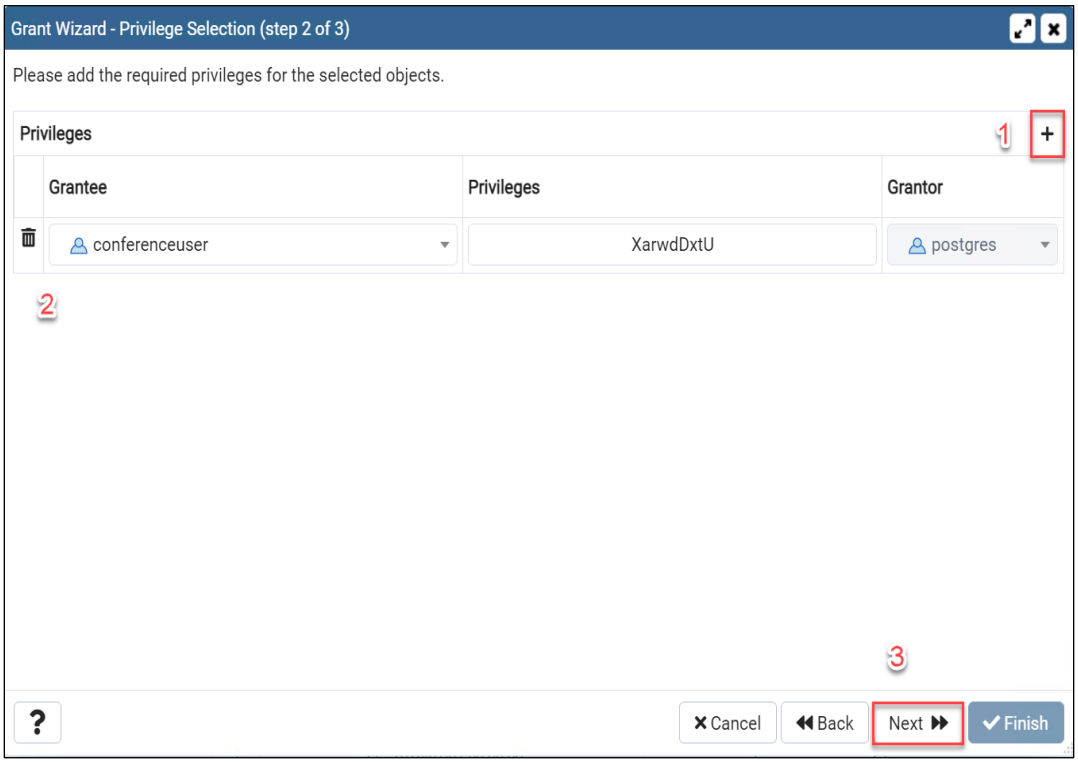
- Open the PostgreSQL pgAdmin tool
  - If opening for the first time, set the admin password to Seattle123
- Expand the **Servers** node
- Expand the **PostgreSQL 10** node. Connect to your local PostgreSQL instance
  - Enter Seattle123 for the password
- Expand the **Databases** node
- Right-click the **Databases** node, select **Create->Database**
- For the name, type **reg_app**, select **Save**
- Expand the **reg_app** node
- Right-click the **Schemas** node, select **Query tool**
- Select the open file icon

- Browse to **C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\database-scripts**
- Select **conferencedemo-PostgreSQL-10.sql** file, select **Select**
- Press **F5** to execute the sql
- Create the database user
  - In the navigator, right-click the **Login/Group Roles**
  - Select **Create->Login/Group Rule**
  - For the name, type **conferenceuser**
  - Select the **Definition** tab, type Seattle123 for the password
  - Select the **Privileges** tab, toggle the **Can login?** to yes
  - Select **Save**
- To allow the database user to perform DML operations against the database, right-click the **reg_app** schema in the **reg_app** database and select **Grant Wizard...**.
- Select all enumerated objects. Then, select **Next**.




This image demonstrates how to use the Grant Wizard to grant permissions over all database objects to conferenceuser.

- Select the + button
- Choose **conferenceuser** as the **Grantee**
- Select **ALL** Privileges and choose **postgres** as the **Grantor**.
- Select **Next**.




Using the Grant Wizard to select what permissions should be granted to the selected objects.

- Inspect the generated SQL code. Select **Finish**.

  **Note**: The permissions granted are more than what is necessary for the app to function.

## Configure Blob Data

- Open the **C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\database-scripts\sample-images** directory
- Copy the image files to the **C:\Program Files\PostgreSQL\10\data** directory
- Right-click the **reg_app** schema, select **Query Tool**
- Browse to the **C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\database-scripts\load-images.sql** file
- Press **F5** to execute it

**Note**: the pg_read_binary_file parameter is a relative path from the PostgreSQL data directory

## Install Azure CLI

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Download and Install the Azure CLI

## Install NodeJS

Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Download and Install NodeJS. Select the LTS 64-bit MSI Installer.
  - Accept the default installation location
  - Make sure that the **Automatically install the necessary tools** box is NOT selected

## Install and Configure Visual Studio Code

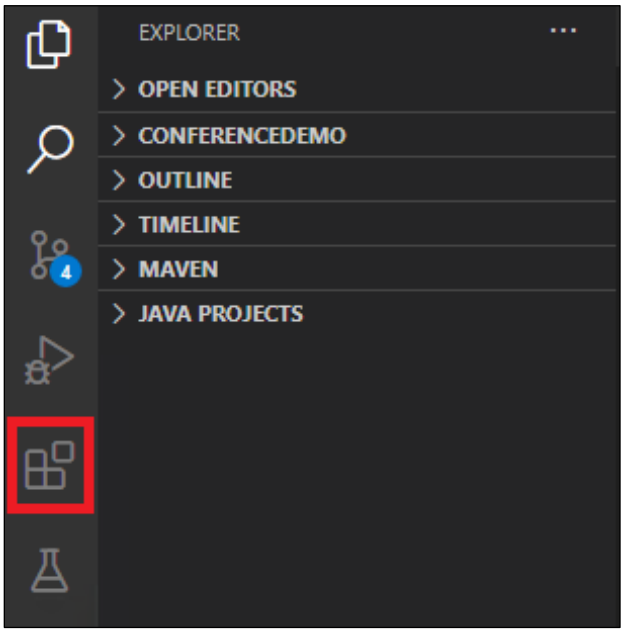
Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Download and Install Visual Studio Code.
- Select the 64-bit Windows User Installer

## Configure the Web Application API

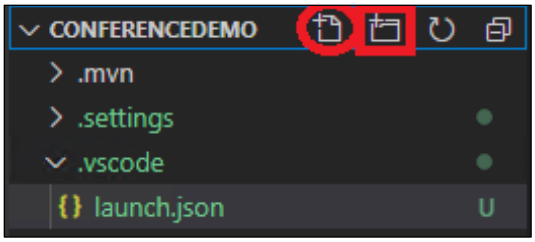
Perform the following on the **PREFIX-vm-pgdb01** virtual machine resource.

- Open Visual Studio Code, if prompted, select **Yes, I trust the authors**
- Open the **C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo** folder (Ctrl+K and Ctrl+O, or **File->Open Folder...**)
- Select the **Extensions** tab




Opening the Visual Studio Code extensions tab.

- Search for and install the following extensions
  - PostgreSQL (by Microsoft)
  - Java Extension Pack
  - Spring Initializer Java Support
- When prompted, select **Yes** to trust the **Maven Wrapper**
- Update the .vscode\launch.json file
  - If a launch.json does not exist, create a .vscode folder, and then create a new file called launch.json. The rectangle highlights the tool used to create a new folder, while the oval indicates the tool to create a new file.



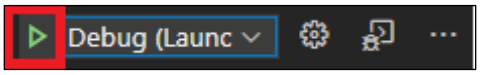

Creating .vscode folder and launch.json file in Visual Studio Code.

  - Copy the following into it:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "java",
            "name": "Debug (Launch)",
            "request": "launch",
            "mainClass": "com.yourcompany.conferencedemo.ConferencedemoApplication",
            "env" :{
                "DB_CONNECTION_URL" : "jdbc:postgresql://localhost:5432/reg_app?useUnicode=true
&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
&noAccessToProcedureBodies=true",
                "DB_USER_NAME" : "conferenceuser",
                "DB_PASSWORD" : "Seattle123",
                "ALLOWED_ORIGINS" : "*",
            }
        }
    ]
}
```

- Update the **{DB_CONNECTION_URL}** environment variable to the PostgreSQL Connections string jdbc:postgresql://localhost:5432/reg_app?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC&noAccessToProcedureBodies=true
- Update the **{DB_USER_NAME}** environment variable to the PostgreSQL Connections string conferenceuser
- Update the **{DB_PASSWORD}** environment variable to the PostgreSQL Connections string Seattle123
- Update the **{ALLOWED_ORIGINS}** environment variable to *

- Select the **Debug** tab (directly above the **Extensions** tab from earlier), then select the debug option to start a debug session




Launching Java API debugging in Visual Studio Code.

- If prompted, select **Yes** to switch to standard mode

## Test the Web Application

- Open a browser window, browse to **http://localhost:8888/api/v1/attendees**
- Ensure the application started on port 8888 and displays results

## Configure the Web Application Client

- Open a new Visual Studio Code window to **C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\testapp\conferencedemo-client**
- Open a terminal window (**Terminal**->**New Terminal**)
- Run the following commands to install all the needed packages, if prompted, select **N**

```
$env:Path = [System.Environment]::GetEnvironmentVariable("Path","Machine")

npm install
npm install -g @angular/cli
```

**Note**: If PowerShell indicates that npm is not a recognized command, try restarting VS Code.

- Close the terminal window and open a new one
- Run the following commands to run the client application

```
ng serve -o
```

- A browser will open to the node site **http://localhost:{port}**
- Browse the conference site, ensure sessions and speaker pages load

**Note**: If you do not see any results, verify that the API is still running.

## Deploy the Java Server Application to Azure

- Open a command prompt window
- Run the following command to create the Maven configuration to deploy the app.
- Be sure to replace the maven version (ex 3.8.1)

```
cd C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo
mvn com.microsoft.azure:azure-webapp-maven-plugin:1.15.0:config
```

- Multiple packages will be installed from the Maven repository. Wait for the process to complete.

- When prompted, for the Define value for OS (Default: Linux), select the option that corresponds to Linux or press **ENTER**
- Select Java 11
- Type **Y** to confirm the settings, then press **ENTER**

```
Please confirm webapp properties
AppName : conferencedemo-1600313725405
ResourceGroup : conferencedemo-1600313725405-rg
Region : westeurope
PricingTier : PremiumV2_P1v2
OS : Linux
RuntimeStack : JAVA 11-java11
Deploy to slot : false
Confirm (Y/N)? : Y
```

Azure Maven plugin initial configuration.

- Switch to Visual Studio Code and the **ConferenceDemo** project
- Open the pom.xml file, notice the **com.microsoft.azure** groupId is now added
- Modify the resource group, appName and region to match the ones deployed in the sample ARM template

  **Note**: You may also need to specify the appServicePlanName and appServicePlanResourceGroup fields in the file, given that the ARM template already deploys an App Service plan. Here is an example of how the pom.xml file looks. We recommend using conferencedemo-api-app-[SUFFIX] as the appName. However, if the ARM template already deploys an App Service instance for you, please use that name instead.

```
<plugin>
  <groupId>com.microsoft.azure</groupId>
  <artifactId>azure-webapp-maven-plugin</artifactId>
  <version>1.15.0</version>
  <configuration>
    <schemaVersion>v2</schemaVersion>
    <subscriptionId>[redacted]</subscriptionId>
    <appServicePlanResourceGroup>[redacted]</appServicePlanResourceGroup>
    <appServicePlanName>skm-pg-appsvc</appServicePlanName>
    <resourceGroup>[redacted]</resourceGroup>
    <appName>conferencedemo-api-app</appName>
    <pricingTier>P1v2</pricingTier>
    <region>eastus2</region>
    <runtime>
      <os>Linux</os>
      <javaVersion>Java 11</javaVersion>
      <webContainer>Java SE</webContainer>
    </runtime>
    <deployment>
      <resources>
        <resource>
          <directory>${project.basedir}/target</directory>
          <includes>
            <include>*.jar</include>
          </includes>
        </resource>
      </resources>
    </deployment>
  </configuration>
</plugin>
```

Azure Webapp Maven Plugin final configuration in pom.xml for Java API app.

- If you have more than one subscription, set the specific subscriptionId in the maven configuration
  - Add the subscriptionId xml element and set to the target subscription
- If the secure landing zone has been deployed, set the hosts file
  - Browse to your resource group, select the **PREFIXapi01** app service
  - Select **Networking**
  - Select **Configure your Private Endpoint connections**

- Select the **PREFIXapi-pe** private endpoint
- Record the private IP Address
- Repeat for the **PREFIXapp01** app service
- Open a Windows PowerShell ISE window
- Copy in the code from below, be sure to replace tokens, and save to **C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts** as **ConfiguringHostsFile.ps1**

```
$prefix = "{PREFIX}";
$apiip = "{APIIP}";
$app_name = "$($prefix)api01";

$hostname = "$app_name.azurewebsites.net"
$line = "$apiip`t$hostname"
add-content "c:\windows\system32\drivers\etc\hosts" $line

$hostname = "$app_name.scm.azurewebsites.net"
$line = "$apiip`t$hostname"
add-content "c:\windows\system32\drivers\etc\hosts" $line

$appip = "{APPIP}"
$app_name = "$($prefix)app01";
$hostname = "$app_name.azurewebsites.net"
$line = "$appip`t$hostname"
add-content "c:\windows\system32\drivers\etc\hosts" $line

$hostname = "$app_name.scm.azurewebsites.net"
$line = "$appip`t$hostname"
add-content "c:\windows\system32\drivers\etc\hosts" $line
```

- Run the file

Running a PowerShell Script for the secure ARM template in PowerShell ISE.

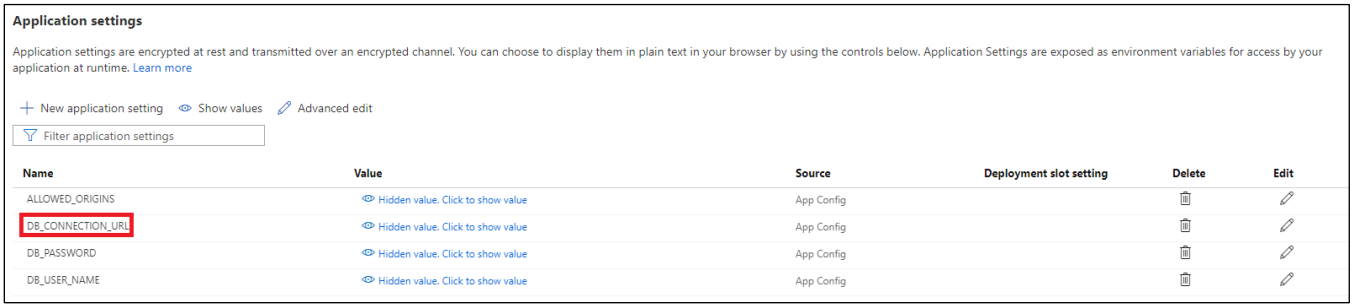
- In the command prompt window from earlier, run the following to deploy the application. Be sure to replace the maven version (ex 3.8.1)

```
mvn package azure-webapp:deploy
```

- When prompted, login to the Azure Portal
- Update the App Service configuration variables by running the following, be sure to replace the tokens:

```
$prefix = "{PREFIX}";
$app_name = "$($prefix)api01";
$rgName = "{RESOURCE-GROUP-NAME}";
az login
az account set --subscription "{SUBSCRIPTION-ID}"
az webapp config appsettings set -g $rgName -n $app_name --settings DB_CONNECTION_URL={DB_CONNECTIO
N_URL}
az webapp config appsettings set -g $rgName -n $app_name --settings DB_USER_NAME={DB_USER_NAME}
az webapp config appsettings set -g $rgName -n $app_name --settings DB_PASSWORD={DB_PASSWORD}
az webapp config appsettings set -g $rgName -n $app_name --settings ALLOWED_ORIGINS=*
```

**Note**: You will need to escape the ampersands in the connection string. You may consider inputting the value through Azure Portal as well. Navigate to the API App Service and select **Configuration** under **Settings**. Then, under **Application settings**, manually enter the values.




Observing the Java application settings in the Azure Portal.

- Restart the Java API App Service by running the following

```
az webapp restart -g $rgName -n $app_name
```

## Deploy the Angular Web Application to Azure

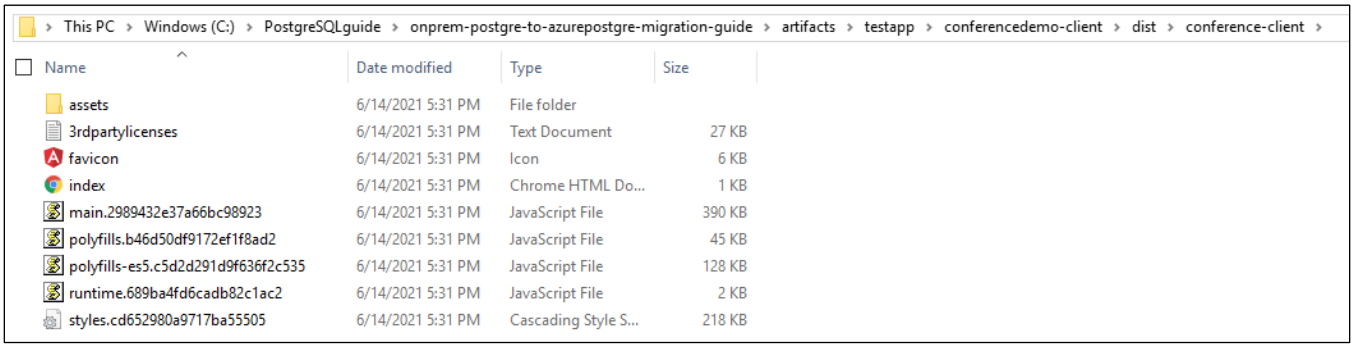
- Switch to the Visual Studio Code window for the Angular app (Conferencedemo-client)
- Navigate to **src\environments\environment.prod.ts**
- Set **webApiUrl** to **[JAVA APP SERVICE URL]/api/v1**

**Note**: the App service url will come from the App Gateway service blade if using the secure deployment, or the App Service blade if not using the secure deployment.

- Run the following command to package the client app

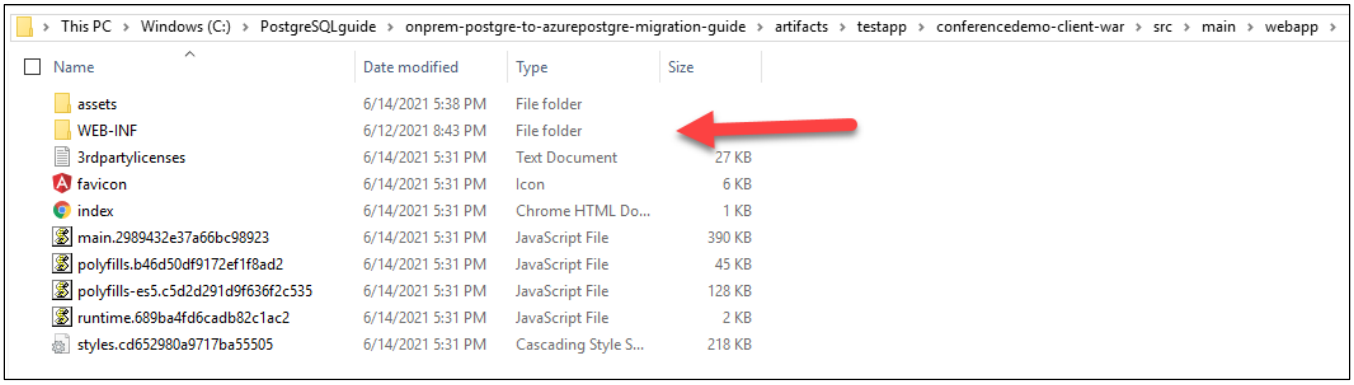
```
ng build --configuration production
```

- Navigate to C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-client\dist\conference-client and copy the contents of that directory




This image demonstrates the artifacts of the production-ready, built Angular app.

- Open a new File Explorer window and navigate to C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-client-war\src\main\webapp. In this case, the built Angular app is packaged as a WAR archive, where it will be served by a Tomcat server in Azure App Service. Paste the contents copied in the previous step to this folder. Be mindful of the existing WEB-INF directory.




The built Angular app is added to the content root path of the WAR archive.

- Use your preferred text editor to open C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-client-war\pom.xml. Take note of the azure-webapp-maven-plugin that has already been added for you.

```
<plugins>
  <plugin>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-webapp-maven-plugin</artifactId>
    <version>1.15.0</version>
    <configuration>
      <schemaVersion>v2</schemaVersion>
      <subscriptionId>[redacted]</subscriptionId>
      <appServicePlanResourceGroup>[redacted]</appServicePlanResourceGroup>
      <appServicePlanName>skm-pg-appsvc</appServicePlanName>
      <resourceGroup>[redacted]</resourceGroup>
      <appName>conferencedemo-client-app</appName>
      <pricingTier>P1v2</pricingTier>
      <region>eastus2</region>
      <runtime>
        <os>Linux</os>
        <javaVersion>Java 11</javaVersion>
        <webContainer>Tomcat 9.0</webContainer>
      </runtime>
      <deployment>
        <resources>
          <resource>
            <directory>${project.basedir}/target</directory>
            <includes>
              <include>*.war</include>
            </includes>
          </resource>
        </resources>
      </deployment>
    </configuration>
  </plugin>
```

azure-webapp-maven-plugin configured in the WAR POM file.

- Ensure that the following fields have been properly completed. Then, in command prompt (or the VS Code terminal), type mvn package azure-webapp:deploy.
  - **subscriptionId**: Your Azure subscription ID.
  - **appServicePlanResourceGroup**: The resource group used to complete this lab.
  - **appServicePlanName**: The App Service plan deployed with the ARM template (i.e. **PLACEHOLDER-pg-appsvc**)
  - **resourceGroup**: The resource group used to complete this lab.

  - **appName**: Provide a unique value, such as **conferencedemo-client-app-SUFFIX**. If the ARM template already deploys an App Service instance intended for the single-tenant Angular client app, use that value instead.

- Once the deployment succeeds, navigate to the sample app in your browser. Confirm that everything works as anticipated.
- Congratulations. You have migrated the sample app to Azure. Now, focus on migrating a multi-tenant app to Azure to explore the power of horizontally scalable PostgreSQL (Citus).

## Configure Network Security (Secure path)

- When attempting to connect to the database from the app service, an access denied message should be displayed. Add the app virtual network to the firewall of the Azure Database for PostgreSQL
  - Browse to the Azure Portal
  - Select the target resource group
  - Select the {PREFIX}PostgreSQL resource
  - Select **Connection security**
  - Select the Allow access to all Azure Services toggle to Yes
  - Select **Save**

# Appendix B: ARM Templates

## Secure

This template will deploy all resources with private endpoints. This effectively removes any access to the PaaS services from the internet.

artifacts\\template-secure.json

## Non-Secure

This template will deploy resources using standard deployment where all resources are available from the internet.

artifacts\template.json

# Appendix B: Citus (Multi-Tenant) App Configuration

This document demonstrates how to develop an application that leverages Azure Database for PostgreSQL (Citus).

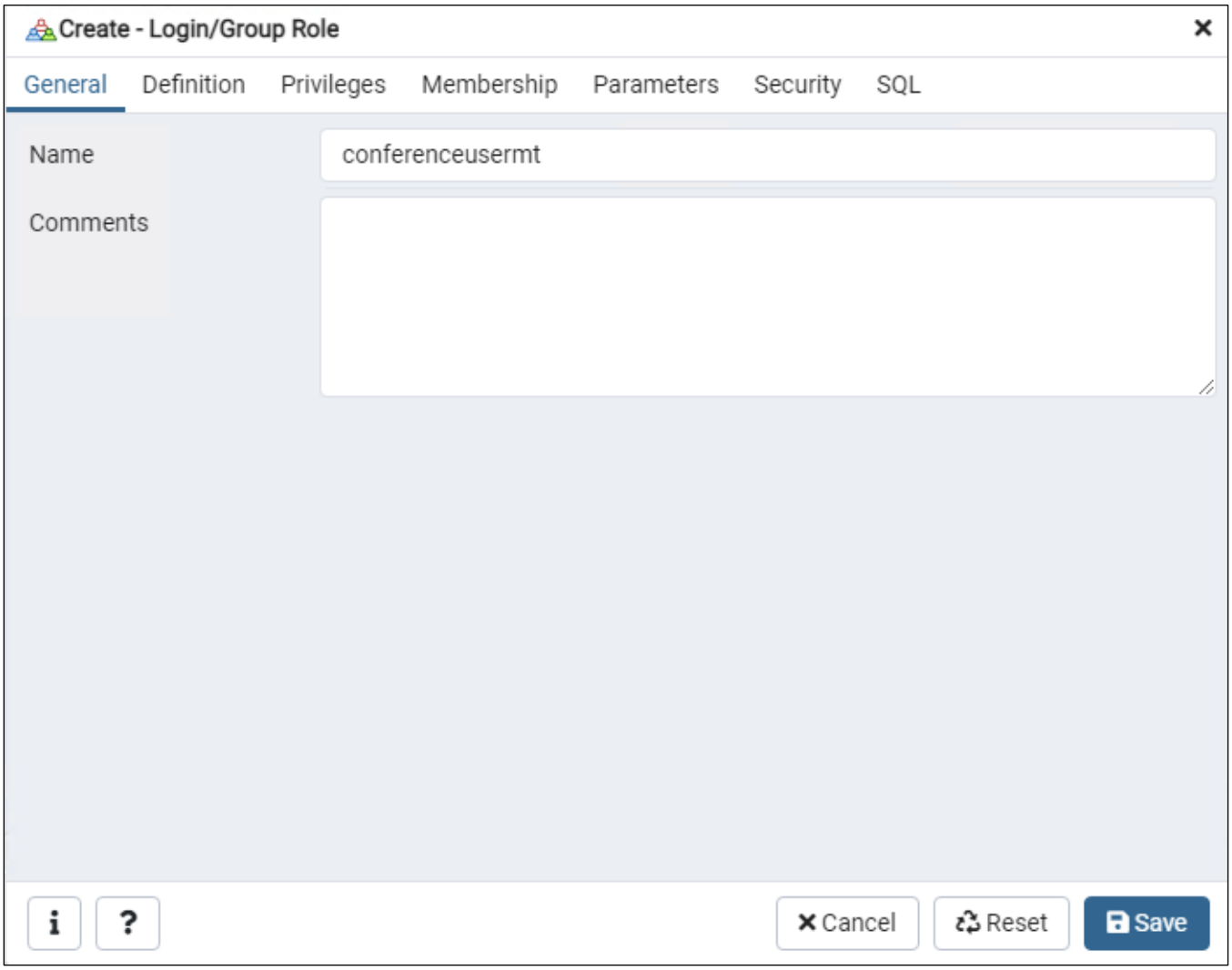
## Setup

We recommend that you follow instructions in the Setup document first. That document will guide you through the process of configuring your VM for local development and deploying a Java API and Angular client app to an Azure landing zone.

Once you complete this, create a database in your local PostgreSQL instance titled reg_app_multitenant. After that, execute the

artifacts\\testapp\\database-scripts\\conferencedemo-postgresql-citus.sql

Now, right-click **Login/Group Roles**. Enter conferenceusermt as the **Name**.




Creating a new user for the multi-tenant app.

Under the **Definition** tab, enter a password of Seattle123. Under the **Privileges** tab, allow the user to login. Disable all other privileges. Then, select **Save**.

Right-click the reg_app_multitenant database and select the **Grant Wizard**.

- Select all objects and select **Next.**
- Configure conferenceusermt as the **Grantee**. Ensure **All** privileges are selected. Then, select **Next**.
- Inspect the generated SQL. Then, select **Finish**.

You are ready to proceed to the next section.

## App Comparison

To understand how the multi-tenant app functions, compare it to the single-tenant app deployed previously.

### Database Changes

The multi-tenant sample app targets Azure Database for PostgreSQL (Citus). Citus instances consist of multiple nodes, which allow for horizontal scaling. However, to applications, the Citus instance appears as one database. This feature allows developers to take advantage of the features of relational databases, while providing the scale needed to support large, multi-tenant applications.

To optimize the performance of this configuration, locate all data belonging to one tenant on a single node. This strategy offers performance benefits (avoiding unnecessary network requests) and allows Citus to enforce constraints, a key advantage of SQL databases.

The modified Citus schema consists of a companies table, and each table used by the application contains a company_id column.

```
-- Not all constraints shown in this code sample

-- Tenants
CREATE TABLE reg_app.companies (
        ID serial PRIMARY KEY,
        Name varchar(50) NOT NULL
);

-- Sessions table used by the application
CREATE TABLE REG_APP.SESSIONS (
   ID INT,
        COMPANY_ID INT,
        NAME VARCHAR(300),
        DESCRIPTION VARCHAR(2000),
        SESSION_DATE TIMESTAMP,
        SPEAKER_ID INT,
        EVENT_ID INT,
        DURATION DECIMAL
);
```

It is relatively easy to account for this new company_id column using the DML UPDATE statement. Each session references an event, hence the reference to the events table.

```
UPDATE REG_APP.SESSIONS
SET COMPANY_ID = events.company_id
FROM reg_app.events
WHERE sessions.event_id = events.id;
```

The code samples indicated above solve the issue of co-location: with the company_id column, Citus easily identifies the owner (tenant) of each row in all application tables in the relational data model. However, to protect referential integrity, Citus requires developers to include the *distribution column* – company_id in this scenario – in composite primary and foreign keys.

```
-- Composite primary key for sessions table
ALTER TABLE REG_APP.SESSIONS ADD PRIMARY KEY (COMPANY_ID, ID);

-- Composite foreign keys for sessions table
ALTER TABLE REG_APP.SESSIONS ADD FOREIGN KEY (COMPANY_ID, EVENT_ID)
   REFERENCES REG_APP.EVENTS (COMPANY_ID, ID);
ALTER TABLE REG_APP.SESSIONS ADD FOREIGN KEY (COMPANY_ID, SPEAKER_ID)
   REFERENCES REG_APP.SPEAKERS (COMPANY_ID, ID);
```

Note that this also applies to UNIQUE constraints. The SQL below does not directly relate to the sessions table, but it enforces that each attendee can enroll for a given session only once.

```
CREATE UNIQUE INDEX IX_REG_SESS_ATTENDEE ON REG_APP.REGISTRATIONS (COMPANY_ID, SESSION_ID, ATT
ENDEE_ID);
```

At the end of this guide, we will use the functions provided by the Citus extension in Azure to support our changes.

### Application Changes

To support the database schema changes, the Java backend has been modified appropriately.

While the Angular frontend has been modified (all URL routes are prefaced with the company ID), that is not the focus of this section.

First, all database requests must be filtered by the company_id column. Hence, each model class has been annotated with the @Filter() Java annotation.

```
@Filter(name = "TenantFilter", condition = "company_id = :tenantId")
```

The filter references a tenantId value. To provide this value on every request to the database, the application leverages AspectJ. The @Before() annotation intercepts method calls to AttendeeService objects. The addFilter() method then appends the tenantId filter parameter to requests.

```
@Before("execution(* com.yourcompany.conferencedemo.services.AttendeeService.*(..)) && target(attendeeServic
e)")
public void addFilter(JoinPoint pjp, AttendeeService attendeeService)
{
   org.hibernate.Filter filter = attendeeService.entityManager.unwrap(Session.class).enableFilter("TenantFilter");
   filter.setParameter("tenantId", TenantContext.getCurrentTenant());
```

```
    filter.validate();
}
```

**Note**: In the multi-tenant app, injected service classes handle database operations for API controllers; the API controllers themselves do not make calls to repository methods.

The API initiates a thread to handle each request. So, the TenantContext class exposes the thread-specific value of the current tenant.

The Java API app fetches records using both their company_id and individual object ID. While it is also possible to fetch a record using its object ID, that does not leverage Citus distribution on the company_id column.

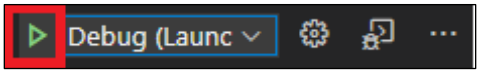
## Run the Sample App Locally

### Run the Java Application

- Open a Visual Studio Code instance to C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-multitenant.
- There is a file called launch.json in the .vscode directory. Ensure that the DB_CONNECTION_URL key of the env object has the following connection string:

  ```
  jdbc:postgresql://localhost:5432/reg_app_multitenant?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC&noAccessToProcedureBodies=true&sslmode=require
  ```

- Verify that the DB_USER_NAME key is set to conferenceusermt
- Verify that the DB_PASSWORD key is set to Seattle123
- Select the **Debug** tab (directly above the **Extensions** tab), then select the debug option to start a debug session

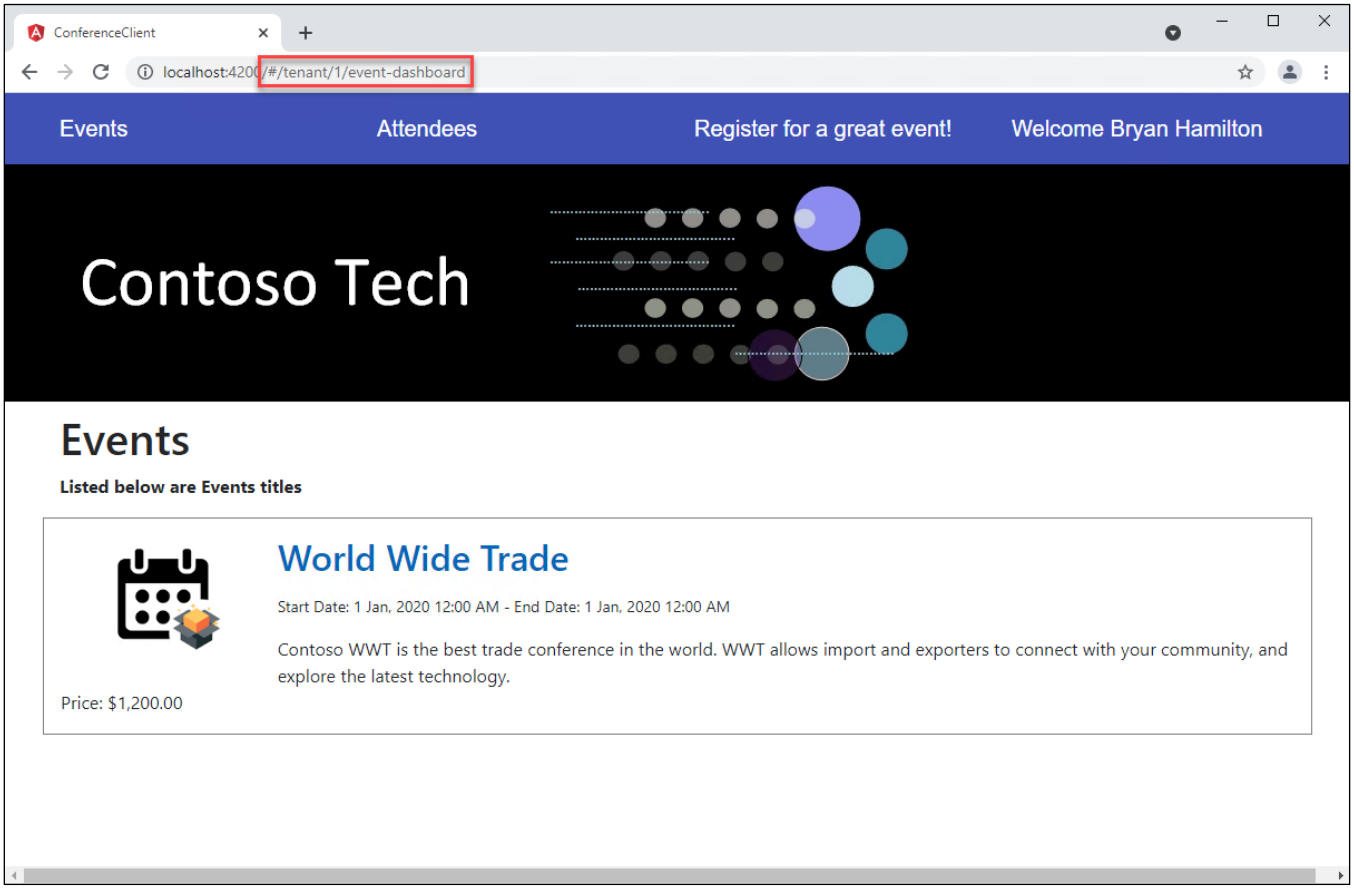
  
  

  Launching Visual Studio Code debugging configuration for Java API app.

- If prompted, select **Yes** to switch to standard mode
- If you face any permissions issues, try the postgres user, or try making the conferenceusermt user a member of the postgres role
- If you'd like to experiment with the API independently of the Angular front-end, you can try using a client such as Postman. Just make sure to append the X-TenantID header to your requests.

## Run the Angular Application

- Navigate to C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-client-multitenant using Visual Studio Code
- Navigate to the environments directory in the src directory. In environment.ts, ensure that webApiUrl references the local API URL (http://localhost:8888/api/v1)
- Using the Visual Studio Code terminal, type ng serve to start the development server. Navigate to localhost:4200. The browser will render the landing page for Tenant 1.




Landing page for Tenant 1, shown in the Chrome browser.

- As this is a multi-tenant application, try the following URLs:
  - http://localhost:4200/#/tenant/2/event-dashboard: should print an event titled "Azure Development"
  - http://localhost:4200/#/tenant/3/event-dashboard: should print an event titled "Machine Learning"

- Feel free to explore the other pages on the sample application by using the navigation bar. In particular, the Attendees page lists the object ID of each registered attendee. You can use this to ensure that the proper tenant displays.

## Data Migration

First, migrate the schema from the source PostgreSQL instance to the Citus instance in Azure. Follow the Data Migration - Schema document and the recommendations it contains.

Then, using either pgAdmin or the psql utility, connect to your Citus instance. Run the following commands.

```
-- Citus does not support distributed tables with triggers
-- See https://github.com/citusdata/citus/issues/906
DROP TRIGGER attendee_insert_trigger ON reg_app.attendees;

-- It does not matter what order you run these commands in
-- Since you dropped all foreign keys
SELECT create_distributed_table('reg_app.companies', 'id');
SELECT create_distributed_table('reg_app.attendees', 'company_id');
SELECT create_distributed_table('reg_app.events', 'company_id');
SELECT create_distributed_table('reg_app.registrations', 'company_id');
SELECT create_distributed_table('reg_app.sessions', 'company_id');
SELECT create_distributed_table('reg_app.speakers', 'company_id');
```

Finally, follow the items contained in the Data Migration to Hyperscale Citus document. Upon completing this, enable foreign key constraints.

**Note**: You may need to execute each ALTER TABLE ... statement individually when adding foreign key constraints.

## More Citus Modifications

Create a Citus *reference table*. Reference tables are just one shard, but that shard is replicated to all worker nodes. In the case of multi-tenant apps, reference tables may contain data that does not belong to a single tenant.

For example, suppose the app is updated to support greater localization. A dropdown may list country names, but the application requires a two-digit country code. This data is not tenant-specific, so it makes sense to configure it as a reference table.

```
CREATE TABLE reg_app.CountryData
(
	ID SERIAL PRIMARY KEY,
	Name varchar(20) NOT NULL,
	CountryCode varchar(5) NOT NULL
);
```

```
INSERT INTO reg_app.countrydata (Name, CountryCode)
VALUES
('United States', 'US'),
('Canada', 'CA');

SELECT create_reference_table('reg_app.countrydata');
```

In addition, in the Citus database, the coordinator node contains metadata regarding the cluster. See this document for more information about Citus' metadata tables.

Considering modern applications more generally, they typically allow their users to optimize the application for their purposes. For example, the Azure portal allows administrators to define custom UI layouts to optimize their experience monitoring their resources.

Suppose the Contoso team wants to provide a simplified version of this feature: they will provide multiple built-in dashboard layouts. One way to include this in the application is through the JSONB datatype, a native feature of PostgreSQL.

Luckily, Citus supports standard local tables. It does not make sense to distribute the UI layout table (it is not tenant-specific configuration, and the data is far too small to benefit), and it does not make sense to make this table a reference table (it will not partake in queries with other application data).

```
CREATE TABLE reg_app.dashboard_layout (
        ID SERIAL PRIMARY KEY,
        dashboard_layout JSONB NOT NULL
);

INSERT INTO reg_app.dashboard_layout (dashboard_layout)
VALUES
('{ "type": "Attendees_Counter", "rowPosition": 2, "colPosition": 1, "width": 1, "height": 1 }'::JSONB),
('{ "type": "Conference_Feedback_Display", "rowPosition": 1, "colPosition": 2, "width": 2, "height": 1 }'::JSONB),
('{ "type": "Sessions_Counter", "rowPosition": 2, "colPosition": 2, "width": 1, "height": 1 }'::JSONB);
```

It is simple to use JSON objects in queries. For example, a dropdown on the frontend may list the names of the layouts:

```
SELECT id, dashboard_layout->>'type' AS LayoutType
FROM reg_app.dashboard_layout;
```

## Migrate App to Azure and Test

This guide details how to migrate both the Java API app and the Angular front-end app to Azure.

## Migrating the Java App

The process of deploying the sample app to Azure is quite simple.

- Navigate to the pom.xml file for the conferencedemo-multitenant Java app. Notice that the azure-webapp-maven-plugin has already been added to the file for you.
- As documented in the Setup, specify the following values in the file:
  - **subscriptionId**: the ID of the Azure subscription you have used in this lab
  - **appServicePlanResourceGroup**: specify the resource group used in the lab
  - **appServicePlanName**: use the format **NAMESPACE-pg-appsvc**
  - **resourceGroup**: specify the resource group used in the lab
  - **appName**: if an App Service instance has not been deployed in the ARM template for the multi-tenant Java API app, use a name such as **conferencedemo-multitenant-api-NAMESPACE**
  - **region**: keep the App Service instance in the same region as your other lab resources
- Once you have configured the pom.xml file, run the following command in the Visual Studio Code terminal to deploy the app to Azure.

```
mvn package azure-webapp:deploy
```

### Change Java App Settings

In its current state, the Java API app deployed to Azure will not function, as it has not been configured.

Again, you can use the Azure CLI. Make sure to replace the token and escape special characters, such as ampersands.

```
$prefix = "{PREFIX}";
$app_name = "$($prefix)api01";
$rgName = "{RESOURCE-GROUP-NAME}";
az login
az account set --subscription "{SUBSCRIPTION-ID}"
az webapp config appsettings set -g $rgName -n $app_name --settings DB_CONNECTION_URL={DB_CONNECTIO
N_URL}
az webapp config appsettings set -g $rgName -n $app_name --settings DB_USER_NAME={DB_USER_NAME}
az webapp config appsettings set -g $rgName -n $app_name --settings DB_PASSWORD={DB_PASSWORD}
az webapp config appsettings set -g $rgName -n $app_name --settings ALLOWED_ORIGINS=*
```

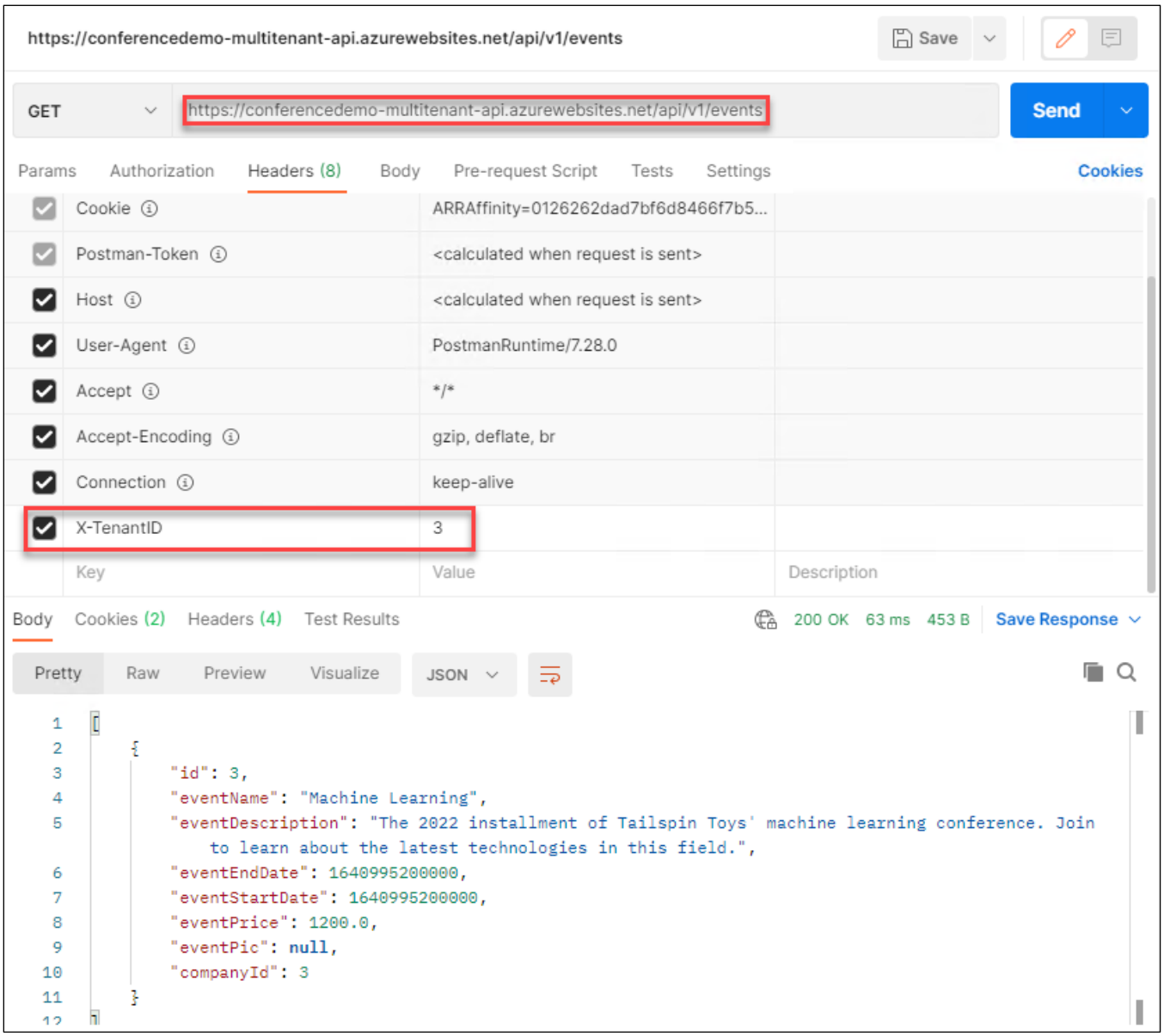
It may be easier to use the Azure portal instead. Configure the follow values as **Application settings**.

- **DB_CONNECTION_URL**: Use the following connection string. Make sure to replace the **NAMESPACE** placeholder.

jdbc:postgresql://c.NAMESPACE-pg-citus-01.postgres.database.azure.com:5432/citus?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC&noAccessToProcedureBodies=true&sslmode=require

- **DB_USER_NAME**: Use **citus** or **conferenceusermt**
- **DB_PASSWORD**: Use **Seattle123Seattle123**
- **ALLOWED_ORIGINS**: Use *****.

Saving the app settings will restart the application. After a couple of minutes, test the API in the client of your choice.



Testing the deployed API app in Postman.

### Deploying the Angular App

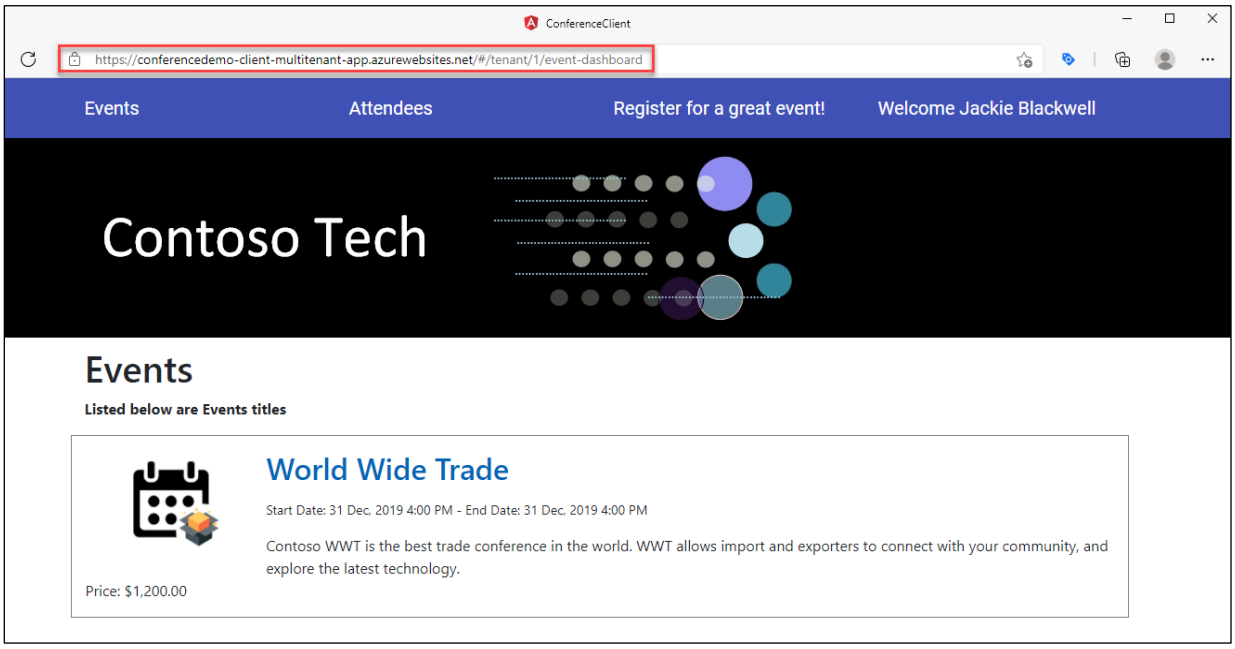
- Navigate to the conferencedemo-client-multitenant project in Visual Studio Code (C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-client-multitenant).
- In the environments directory in the src directory, navigate to environment.prod.ts. Ensure that webApiUrl points to the Java multi-tenant API app that you just deployed
- If so, create a production build of the app by entering the following in the Visual Studio Code terminal:

```
ng build --configuration production
```

- Copy the contents of the generated conference-client folder in the dist directory (same process as the single-tenant app deployment; refer to the Setup document again if you are unclear on the process)
- Navigate to the conferencedemo-client-multitenant-war directory (C:\PostgreSQLguide\onprem-postgre-to-azurepostgre-migration-guide\artifacts\testapp\conferencedemo-client-multitenant-war)
- Paste the copied contents into the src\main\webapp directory, making sure to keep the WEB-INF directory intact
- Populate the pom.xml file with the same set of parameters as the Java API deployments and the single-tenant Angular app deployment:
  - **subscriptionId**
  - **appServicePlanResourceGroup**
  - **appServicePlanName**
  - **resourceGroup**
  - **appName**: If an App Service instance has not been deployed with the ARM template, use a name such as conferencedemo-client-multitenant-app-NAMESPACE
  - **region**
- Once you populate these parameters, run the following command in the Visual Studio Code terminal:

```
mvn package azure-webapp:deploy
```

- Allow the deployment to complete. Test the deployed app in the browser. Everything should function as expected.




Conference Demo multi-tenant client app deployed to Azure.

Congratulations. You have successfully migrated a multi-tenant application to Azure. You have also seen the features of Azure Database for PostgreSQL (Citus) that make it an ideal choice for many SaaS workloads.

# Appendix C: Default server parameters PostgreSQL 10.0

| name | setting | enumvals |
|---|---|---|
| **allow_system_table_mods** | off | |
| **application_name** | psql | |
| **archive_command** | (disabled) | |
| **archive_mode** | off | {always,on,off} |
| **archive_timeout** | 0 | |
| **array_nulls** | on | |
| **authentication_timeout** | 60 | |
| **autovacuum** | on | |
| **autovacuum_analyze_scale_factor** | 0.1 | |

| | | |
|---|---|---|
| **autovacuum_analyze_threshold** | 50 | |
| **autovacuum_freeze_max_age** | 200000000 | |
| **autovacuum_max_workers** | 3 | |
| **autovacuum_multixact_freeze_max_age** | 400000000 | |
| **autovacuum_naptime** | 60 | |
| **autovacuum_vacuum_cost_delay** | 20 | |
| **autovacuum_vacuum_cost_limit** | -1 | |
| **autovacuum_vacuum_scale_factor** | 0.2 | |
| **autovacuum_vacuum_threshold** | 50 | |
| **autovacuum_work_mem** | -1 | |
| **backend_flush_after** | 0 | |
| **backslash_quote** | safe_encoding | {safe_encoding,on,off} |
| **bgwriter_delay** | 200 | |
| **bgwriter_flush_after** | 0 | |
| **bgwriter_lru_maxpages** | 100 | |
| **bgwriter_lru_multiplier** | 2 | |
| **block_size** | 8192 | |
| **bonjour** | off | |
| **bonjour_name** | | |
| **bytea_output** | hex | {escape,hex} |
| **check_function_bodies** | on | |
| **checkpoint_completion_target** | 0.5 | |
| **checkpoint_flush_after** | 0 | |
| **checkpoint_timeout** | 300 | |

| | | |
|---|---|---|
| **checkpoint_warning** | 30 | |
| **client_encoding** | WIN1252 | |
| **client_min_messages** | notice | {debug5,debug4,debug3,debug2,debug1,log,notice,warning,error} |
| **cluster_name** | | |
| **commit_delay** | 0 | |
| **commit_siblings** | 5 | |
| **config_file** | C:/Program Files/PostgreSQL/10/data/postgresql.conf | |
| **constraint_exclusion** | partition | {partition,on,off} |
| **cpu_index_tuple_cost** | 0.005 | |
| **cpu_operator_cost** | 0.0025 | |
| **cpu_tuple_cost** | 0.01 | |
| **cursor_tuple_fraction** | 0.1 | |
| **data_checksums** | off | |
| **data_directory** | C:/Program Files/PostgreSQL/10/data | |
| **data_sync_retry** | off | |
| **DateStyle** | ISO, MDY | |
| **db_user_namespace** | off | |
| **deadlock_timeout** | 1000 | |
| **debug_assertions** | off | |
| **debug_pretty_print** | on | |
| **debug_print_parse** | off | |

| | | |
|---|---|---|
| **debug_print_plan** | off | |
| **debug_print_rewritten** | off | |
| **default_statistics_target** | 100 | |
| **default_tablespace** | | |
| **default_text_search_config** | pg_catalog.english | |
| **default_transaction_deferrable** | off | |
| **default_transaction_isolation** | read committed | {serializable,"repeatable read","read committed","read uncommitted"} |
| **default_transaction_read_only** | off | |
| **default_with_oids** | off | |
| **dynamic_library_path** | $libdir | |
| **segment_size** | 131072 | |
| **seq_page_cost** | 1 | |
| **server_encoding** | UTF8 | |
| **server_version** | 10.16 | |
| **server_version_num** | 100016 | |
| **session_preload_libraries** | | |
| **session_replication_role** | origin | {origin,replica,local} |
| **shared_buffers** | 16384 | |
| **shared_preload_libraries** | | |
| **ssl** | off | |
| **ssl_ca_file** | | |
| **ssl_cert_file** | server.crt | |
| **ssl_ciphers** | HIGH:MEDIUM:+3DES:!aNULL | |

| | | |
|---|---|---|
| **ssl_crl_file** | | |
| **ssl_dh_params_file** | | |
| **ssl_ecdh_curve** | prime256v1 | |
| **ssl_key_file** | server.key | |
| **ssl_prefer_server_ciphers** | on | |
| **standard_conforming_strings** | on | |
| **statement_timeout** | 0 | |
| **stats_temp_directory** | pg_stat_tmp | |
| **superuser_reserved_connections** | 3 | |
| **synchronize_seqscans** | on | |
| **synchronous_commit** | on | {local,remote_write,remote_apply,on,off} |
| **synchronous_standby_names** | | |
| **syslog_facility** | none | {none} |
| **syslog_ident** | postgres | |
| **syslog_sequence_numbers** | on | |
| **syslog_split_messages** | on | |
| **tcp_keepalives_count** | 0 | |
| **tcp_keepalives_idle** | -1 | |
| **tcp_keepalives_interval** | -1 | |
| **temp_buffers** | 1024 | |
| **temp_file_limit** | -1 | |
| **temp_tablespaces** | | |
| **TimeZone** | UTC | |
| **timezone_abbreviations** | Default | |

| | | |
|---|---|---|
| **trace_notify** | off | |
| **trace_recovery_messages** | log | {debug5,debug4,debug3,debug2,debug1,log,notice,warning,error} |
| **trace_sort** | off | |
| **track_activities** | on | |
| **track_activity_query_size** | 1024 | |
| **track_commit_timestamp** | off | |
| **track_counts** | on | |
| **track_functions** | none | {none,pl,all} |
| **track_io_timing** | off | |
| **transaction_deferrable** | off | |
| **transaction_isolation** | read committed | |
| **transaction_read_only** | off | |
| **transform_null_equals** | off | |
| **unix_socket_directories** | | |
| **unix_socket_group** | | |
| **unix_socket_permissions** | 0777 | |
| **update_process_title** | off | |
| **vacuum_cost_delay** | 0 | |
| **vacuum_cost_limit** | 200 | |
| **vacuum_cost_page_dirty** | 20 | |
| **vacuum_cost_page_hit** | 1 | |
| **vacuum_cost_page_miss** | 10 | |
| **vacuum_defer_cleanup_age** | 0 | |
| **vacuum_freeze_min_age** | 50000000 | |

| | | |
|---|---|---|
| **vacuum_freeze_table_age** | 150000000 | |
| **vacuum_multixact_freeze_min_age** | 5000000 | |
| **vacuum_multixact_freeze_table_age** | 150000000 | |
| **wal_block_size** | 8192 | |
| **wal_buffers** | 512 | |
| **wal_compression** | off | |
| **wal_consistency_checking** | | |
| **wal_keep_segments** | 0 | |
| **wal_level** | replica | {minimal,replica,logical} |
| **wal_log_hints** | off | |
| **wal_receiver_status_interval** | 10 | |
| **wal_receiver_timeout** | 60000 | |
| **wal_retrieve_retry_interval** | 5000 | |
| **wal_segment_size** | 2048 | |
| **wal_sender_timeout** | 60000 | |
| **wal_sync_method** | open_datasync | {fsync,fsync_writethrough,open_datasync} |
| **wal_writer_delay** | 200 | |
| **wal_writer_flush_after** | 128 | |
| **work_mem** | 4096 | |
| **xmlbinary** | base64 | {base64,hex} |
| **xmloption** | content | {content,document} |
| **zero_damaged_pages** | off | |

# Appendix D: Default server parameters Azure (Single Server) V11

| Name | Value | AllowedValue |
|---|---|---|
| **array_nulls** | on | on,off |
| **autovacuum** | on | on,off |
| **autovacuum_analyze_scale_factor** | 0.05 | 0-100 |
| **autovacuum_analyze_threshold** | 50 | 0-2147483647 |
| **autovacuum_freeze_max_age** | 200000000 | 100000-2000000000 |
| **autovacuum_max_workers** | 3 | 1-262143 |
| **autovacuum_multixact_freeze_max_age** | 400000000 | 10000-2000000000 |
| **autovacuum_naptime** | 15 | 1-2147483 |
| **autovacuum_vacuum_cost_delay** | 20 | -1-100 |
| **autovacuum_vacuum_cost_limit** | -1 | -1-10000 |
| **autovacuum_vacuum_scale_factor** | 0.05 | 0-100 |
| **autovacuum_vacuum_threshold** | 50 | 0-2147483647 |
| **autovacuum_work_mem** | -1 | -1-2097151 |
| **azure.replication_support** | REPLICA | OFF,REPLICA,LOGICAL |
| **backend_flush_after** | 0 | 0-256 |
| **backslash_quote** | safe_encoding | safe_encoding,on,off |
| **bgwriter_delay** | 20 | 10-10000 |
| **bgwriter_flush_after** | 64 | 0-256 |
| **bgwriter_lru_maxpages** | 100 | 0-1073741823 |
| **bgwriter_lru_multiplier** | 2 | 0-10 |

| | | |
|---|---|---|
| **bytea_output** | hex | escape,hex |
| **check_function_bodies** | on | on,off |
| **checkpoint_completion_target** | 0.9 | 0-1 |
| **checkpoint_warning** | 30 | 0-2147483647 |
| **client_encoding** | sql_ascii | BIG5,EUC_CN,EUC_JP,EUC_JIS_2004,EUC_KR,EUC_TW,GB18 |
| **client_min_messages** | notice | debug5,debug4,debug3,debug2,debug1,log,notice,warn |
| **commit_delay** | 0 | 0-100000 |
| **commit_siblings** | 5 | 0-1000 |
| **connection_throttling** | on | on,off |
| **constraint_exclusion** | partition | partition,on,off |
| **cpu_index_tuple_cost** | 0.005 | 0-1.79769e+308 |
| **cpu_operator_cost** | 0.0025 | 0-1.79769e+308 |
| **cpu_tuple_cost** | 0.01 | 0-1.79769e+308 |
| **cursor_tuple_fraction** | 0.1 | 0-1 |
| **datestyle** | iso, mdy | (iso |
| **deadlock_timeout** | 1000 | 1-2147483647 |
| **debug_print_parse** | off | on,off |
| **debug_print_plan** | off | on,off |
| **debug_print_rewritten** | off | on,off |
| **default_statistics_target** | 100 | 1-10000 |
| **default_text_search_config** | pg_catalog.english | [A-Za-z._]+ |
| **default_transaction_deferrable** | off | on,off |
| **default_transaction_isolation** | read committed | serializable,repeatable read,read committed,read u |

| | | |
|---|---|---|
| **default_transaction_read_only** | off | on,off |
| **default_with_oids** | off | on,off |
| **effective_cache_size** | 655360 | 1-2147483647 |
| **enable_bitmapscan** | on | on,off |
| **enable_hashagg** | on | on,off |
| **enable_hashjoin** | on | on,off |
| **enable_indexonlyscan** | on | on,off |
| **enable_indexscan** | on | on,off |
| **enable_material** | on | on,off |
| **enable_mergejoin** | on | on,off |
| **enable_nestloop** | on | on,off |
| **enable_partitionwise_aggregate** | off | on,off |
| **enable_partitionwise_join** | off | on,off |
| **enable_seqscan** | on | on,off |
| **enable_sort** | on | on,off |
| **enable_tidscan** | on | on,off |
| **escape_string_warning** | on | on,off |
| **exit_on_error** | off | on,off |
| **extra_float_digits** | 0 | -15-3 |
| **force_parallel_mode** | off | off,on,regress |
| **from_collapse_limit** | 8 | 1-2147483647 |
| **geqo** | on | on,off |
| **geqo_effort** | 5 | 1-10 |
| **geqo_generations** | 0 | 0-2147483647 |

| | | |
|---|---|---|
| **geqo_pool_size** | 0 | 0-2147483647 |
| **geqo_seed** | 0.0 | 0-1 |
| **geqo_selection_bias** | 2.0 | 1.5-2 |
| **geqo_threshold** | 12 | 2-2147483647 |
| **gin_fuzzy_search_limit** | 0 | 0-2147483647 |
| **gin_pending_list_limit** | 4096 | 64-2097151 |
| **hot_standby_feedback** | on | on,off |
| **idle_in_transaction_session_timeout** | 0 | 0-2147483647 |
| **intervalstyle** | postgres | postgres,postgres_verbose,sql_standard,iso_8601 |
| **join_collapse_limit** | 8 | 1-2147483647 |
| **lc_monetary** | English_United States.1252 | [A-Za-z0-9._]+ |
| **lc_numeric** | English_United States.1252 | [A-Za-z0-9._]+ |
| **lo_compat_privileges** | off | on,off |
| **lock_timeout** | 0 | 0-2147483647 |
| **log_autovacuum_min_duration** | -1 | -1-2147483647 |
| **log_checkpoints** | on | on,off |
| **log_connections** | on | on,off |
| **log_disconnections** | off | on,off |
| **log_duration** | off | on,off |
| **log_error_verbosity** | default | terse,default,verbose |
| **log_line_prefix** | %t-%c- | .* |
| **log_lock_waits** | off | on,off |
| **log_min_duration_statement** | -1 | -1-2147483647 |

| | | |
|---|---|---|
| **log_min_error_statement** | error | debug5,debug4,debug3,debug2,debug1,info,notice,war |
| **log_min_messages** | warning | debug5,debug4,debug3,debug2,debug1,info,notice,war |
| **log_replication_commands** | off | on,off |
| **log_retention_days** | 3 | 1-7 |
| **log_statement** | none | none,ddl,mod,all |
| **log_statement_stats** | off | on,off |
| **log_temp_files** | -1 | -1-2147483647 |
| **logging_collector** | on | on,off |
| **maintenance_work_mem** | 131072 | 1024-2097151 |
| **max_locks_per_transaction** | 64 | 10-2147483647 |
| **max_parallel_workers** | 8 | 0-1024 |
| **max_parallel_workers_per_gather** | 2 | 0-1024 |
| **max_prepared_transactions** | 0 | 0-8388607 |
| **max_replication_slots** | 10 | 10-25 |
| **max_standby_archive_delay** | 30000 | -1-2147483647 |
| **max_standby_streaming_delay** | 30000 | -1-2147483647 |
| **max_wal_senders** | 10 | 10-30 |
| **max_wal_size** | 1024 | 2-2097151 |
| **min_parallel_index_scan_size** | 524288 | 0-715827882 |
| **min_parallel_table_scan_size** | 8388608 | 0-715827882 |
| **min_wal_size** | 256 | 2-2097151 |
| **old_snapshot_threshold** | -1 | -1-86400 |
| **operator_precedence_warning** | off | on,off |

| | | |
|---|---|---|
| **parallel_leader_participation** | on | on,off |
| **parallel_setup_cost** | 1000 | 0-1.79769e+308 |
| **parallel_tuple_cost** | 0.1 | 0-1.79769e+308 |
| **pg_qs.interval_length_minutes** | 15 | 1-30 |
| **pg_qs.max_query_text_length** | 6000 | 100-10000 |
| **pg_qs.query_capture_mode** | none | all,top,none |
| **pg_qs.replace_parameter_placeholders** | off | on,off |
| **pg_qs.retention_period_in_days** | 7 | 1-30 |
| **pg_qs.track_utility** | on | on,off |
| **pg_stat_statements.max** | 5000 | 100-2147483647 |
| **pg_stat_statements.save** | on | on,off |
| **pg_stat_statements.track** | none | top,all,none |
| **pg_stat_statements.track_utility** | on | on,off |
| **pgms_wait_sampling.history_period** | 100 | 1-600000 |
| **pgms_wait_sampling.query_capture_mode** | none | all,none |
| **postgis.gdal_enabled_drivers** | DISABLE_ALL | DISABLE_ALL,ENABLE_ALL |
| **quote_all_identifiers** | off | on,off |
| **random_page_cost** | 4.0 | 0-1.79769e+308 |
| **row_security** | on | on,off |
| **search_path** | "$user", public \|[A-Za-z."$, ]+ | |
| **seq_page_cost** | 1.0 | 0-1.79769e+308 |
| **session_replication_role** | origin | origin,replica,local |
| **shared_preload_libraries** | | ,auto_explain,pgaudit,pg_prewarm,timescaledb |

| | | |
|---|---|---|
| **statement_timeout** | 0 | 0-2147483647 |
| **synchronize_seqscans** | on | on,off |
| **synchronous_commit** | on | local,remote_write,on,off |
| **tcp_keepalives_count** | 0 | 0-2147483647 |
| **tcp_keepalives_idle** | 0 | 0-2147483647 |
| **tcp_keepalives_interval** | 0 | 0-2147483647 |
| **temp_buffers** | 8192 | 8192-16384 |
| **timezone** | UTC | [A-Za-z0-9/+-_]+ |
| **track_activities** | on | on,off |
| **track_activity_query_size** | 1024 | 100-102400 |
| **track_commit_timestamp** | off | on,off |
| **track_counts** | on | on,off |
| **track_functions** | none | none,pl,all |
| **track_io_timing** | on | on,off |
| **transform_null_equals** | off | on,off |
| **vacuum_cost_delay** | 0 | 0-100 |
| **vacuum_cost_limit** | 200 | 1-10000 |
| **vacuum_cost_page_dirty** | 20 | 0-10000 |
| **vacuum_cost_page_hit** | 1 | 0-10000 |
| **vacuum_cost_page_miss** | 10 | 0-10000 |
| **vacuum_defer_cleanup_age** | 0 | 0-1000000 |
| **vacuum_freeze_min_age** | 50000000 | 0-1000000000 |
| **vacuum_freeze_table_age** | 150000000 | 0-2000000000 |
| **vacuum_multixact_freeze_min_age** | 5000000 | 0-1000000000 |

| | | |
|---|---|---|
| **vacuum_multixact_freeze_table_age** | 150000000 | 0-2000000000 |
| **wal_buffers** | 8192 | -1-262143 |
| **wal_receiver_status_interval** | 10 | 0-2147483 |
| **wal_writer_delay** | 200 | 1-10000 |
| **wal_writer_flush_after** | 128 | 0-2147483647 |
| **work_mem** | 4096 | 4096-2097151 |
| **xmlbinary** | base64 | base64,hex |
| **xmloption** | content | content,document |

# Configure PostgreSQL SSL

## Install OpenSSL

- Download the latest version of OpenSSL from here
- Run the MSI installer (ex Win64 OpenSSL v1.1.1k), click through all the dialogs
- Update the PATH environment variable to have the openssl install directory (similar to C:\Program Files\OpenSSL-Win64\bin)
- Update the PATH environment variable to have the openssl install directory (similar to C:\Program Files\OpenSSL-Win64). This is needed for PgBouncer
- Update the OPENSSL_CONF environment variable to be C:\Program Files\OpenSSL-Win64\bin\openssl.cfg

## Create a server certificate

- Open a new command prompt
- Create the server certificates by running the following (from the c:\temp directory):

```
cd c:\temp
openssl genrsa -des3 -out server.temp.key 2048
```

- Enter a pass key (ex Seattle123)
- Run the following commands

```
openssl rsa -in server.temp.key -out server.key
```

- Re-enter the pass key
- Run the following commands

```
openssl req -new -key server.key -out server.csr
```

- Enter in random information for the certificate request

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

- For the third command, you will be asked to enter information for the certificate. Optionally, leave most of the fields blank (with a period), except the Common Name. This self-signed certificate is acceptable because this is not a production deployment. Populate the Common Name with the hostname of the virtual machine, such as [SUFFIX]-vm-pg-db-01.

```
c:\temp\Certs>openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:Los Angeles
Organization Name (eg, company) [Internet Widgits Pty Ltd]:World Wide Importers
Organizational Unit Name (eg, section) []:Importers
Common Name (e.g. server FQDN or YOUR name) []:skm-vm-pg-db-01
Email Address []:test@wwi.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:WWI

c:\temp\Certs>openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
Signature ok
subject=C = US, ST = California, L = Los Angeles, O = World Wide Importers, OU = Importers, CN = skm-vm-pg-db-01, emailAddress = test@wwi.com
Getting Private key
```

- Copy the server.crt and server.key to the PostgreSQL data directory (ex C:\Program Files\PostgreSQL\10\data).

## Configure the pg_hba.conf file

- Open the C:\Program Files\PostgreSQL\10\data\postgresql.conf file
- Uncomment and set the following parameters:

```
ssl=on
...
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
```

- Open the C:\Program Files\PostgreSQL\10\data\pg_hba.conf file
- Copy and then modify the host option to add the hostssl option:

```
hostssl    all    all   0.0.0.0/0 md5
```

**Note**: Optionally, be more specific by only putting the IP of the Azure Database for PostgreSQL instance or connecting tool. This step would have been completed as part of the SSL configuration in the DMS migration path.

## Restart the server

- Run the following commands:

```
net stop postgresql-x64-10
net start postgresql-x64-10
```

The PostgreSQL database should now accept SSL connections.

# Configure PgBouncer

## Create SSL Certs

- Perform the activities in the Configure PostgreSQL SSL.

## Install PgBouncer

It is easiest to utilize the StackBuilder version that comes with the PostgreSQL download to run PgBouncer. It can also be manually downloaded. However, getting PgBouncer to compile on Windows is quite the chore. The StackBuilder install comes with all the pre-compiled dlls and libraries that are needed, whereas the manual download of PgBouncer will require many extra steps.

### Enterprise Database StackBuilder Install

- After running the setup steps, StackBuilder should have downloaded PgBouncer already.
- After setting up SSL, run the edb_pgbouncer.exe installer
- Click through all dialogs
- Open a command prompt, start pgbouncer as a windows service:

```
cd c:\program files (x86)\pgbouncer\bin

pgbouncer -regservice "c:\program files (x86)\pgbouncer\share\pgbouncer.ini"

net start pgbouncer
```

### Fresh Download of PgBounder (Optional)

This path assumed the installer is not available and PgBouncer must be compiled from source code. It can be skipped if the above steps were completed.

- Open the pgbounder github
- Download the latest windows i686.zip
- Unzip the file to c:\program files (x86)\pgbouncer

### Download libevent

- Open the libevent github
- Download the latest release
- Unzip to c:\program files (x86)\libevent

- Compile the libevent dlls
- Copy the libevent dlls to the c:\program files (x86)\pgbouncer directory

**Configure Windows Server**

- Open the C:\OpenSSL-Win64 directory, copy the following files to the c:\program files (x86)\pgbouncer directory
- Remove the x64 from the file names

**Start PgBouncer**

- Open a command prompt as administrator
- Run the following commands:

```
cd c:\program files (x86)\pgbouncer

pgbouncer -regservice pgbouncer.ini

regsvr32 pgbevent.dll

net start pgbouncer
```

## Configure pgbouncer.ini file

- After installing PgBouncer from one of the above paths, open the c:\program files (x86)\pgbouncer\share\pgbouncer.ini file
- Add the following under the [databases] section:

```
reg_app = dbname=reg_app host=localhost port=5432
```

- Scroll down to the TLS settings, modify the following values:

```
;; disable, allow, require, verify-ca, verify-full
client_tls_sslmode = allow

;; Path to file that contains trusted CA certs
client_tls_ca_file = C:\Program Files (x86)\PgBouncer\share\root.crt

;; Private key and cert to present to clients.
;; Required for accepting TLS connections from clients.
client_tls_key_file = C:\Program Files (x86)\PgBouncer\share\server.key
client_tls_cert_file = C:\Program Files (x86)\PgBouncer\share\server.crt
```

```
;; fast, normal, secure, legacy, <ciphersuite string>
client_tls_ciphers = normal
```

- Copy the certificates created in the SSL steps above to the c:\program files (x86)\PgBouncer\share folder
- Copy the server.crt and rename the copy to root.crt
- Restart the PgBouncer Service

```
net stop pgbouncer
net start pgbouncer
```

## Test PgBouncer

- With PgBouncer running, switch to pgAdmin
- Right-click the **Servers** node
- Select **Create->Server**
- For the name, type **pgbouncer-localhost**
- Select the **Connection** tab
- For the host name, type **localhost**
- For the port, type **6432**
- Type the password for the **postgres** user
- Click **Save**
- Expand the Databases node, and the reg_app database should be displayed