Running PostgreSQL Database in a Cloud Native Environment with Kubernetes

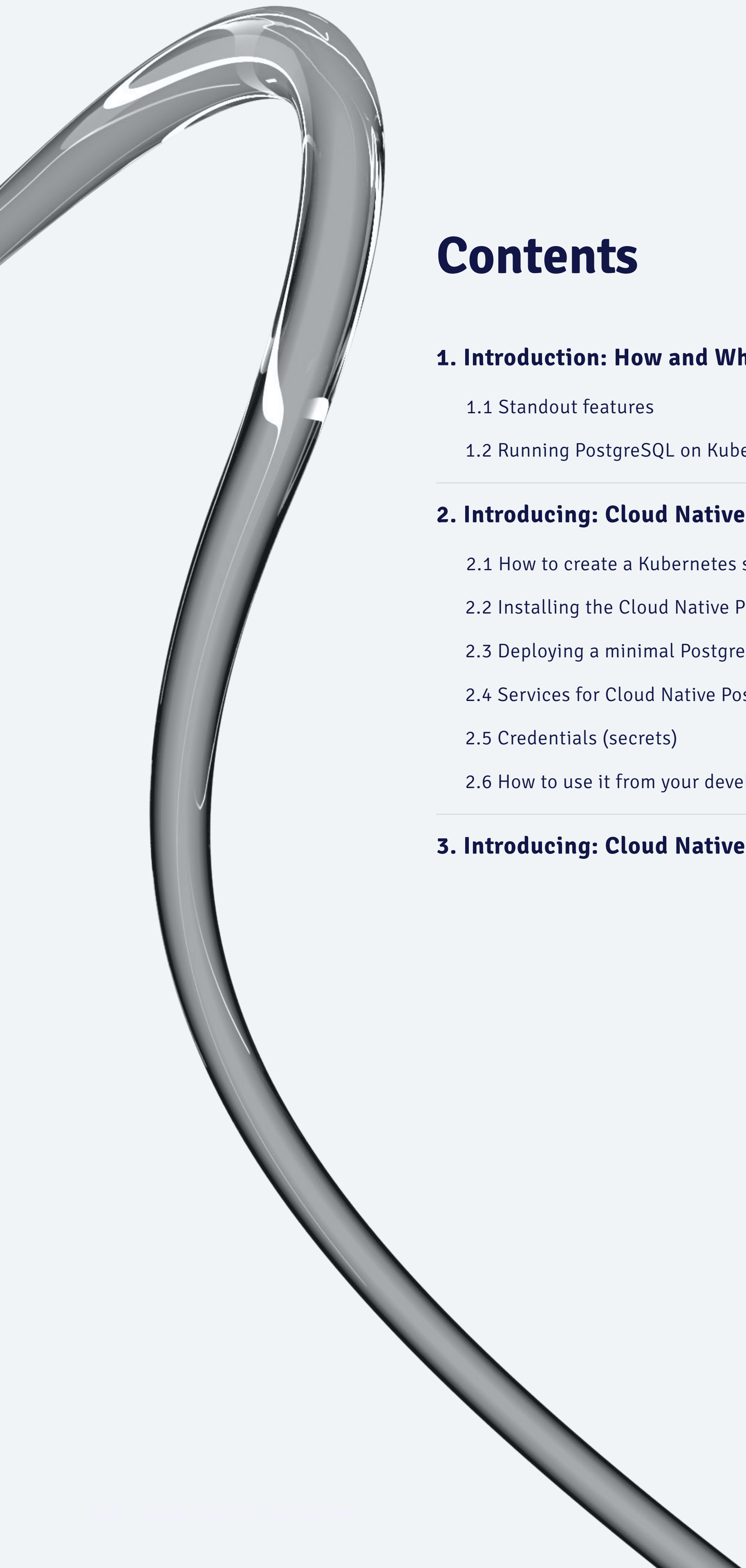
AUTHORED BY:

Gabriele Bartolini
VP, Cloud Native

Leonardo Cecchi
Cloud Native PostgreSQL Lead Developer

POWER TO POSTGRES

EDB™

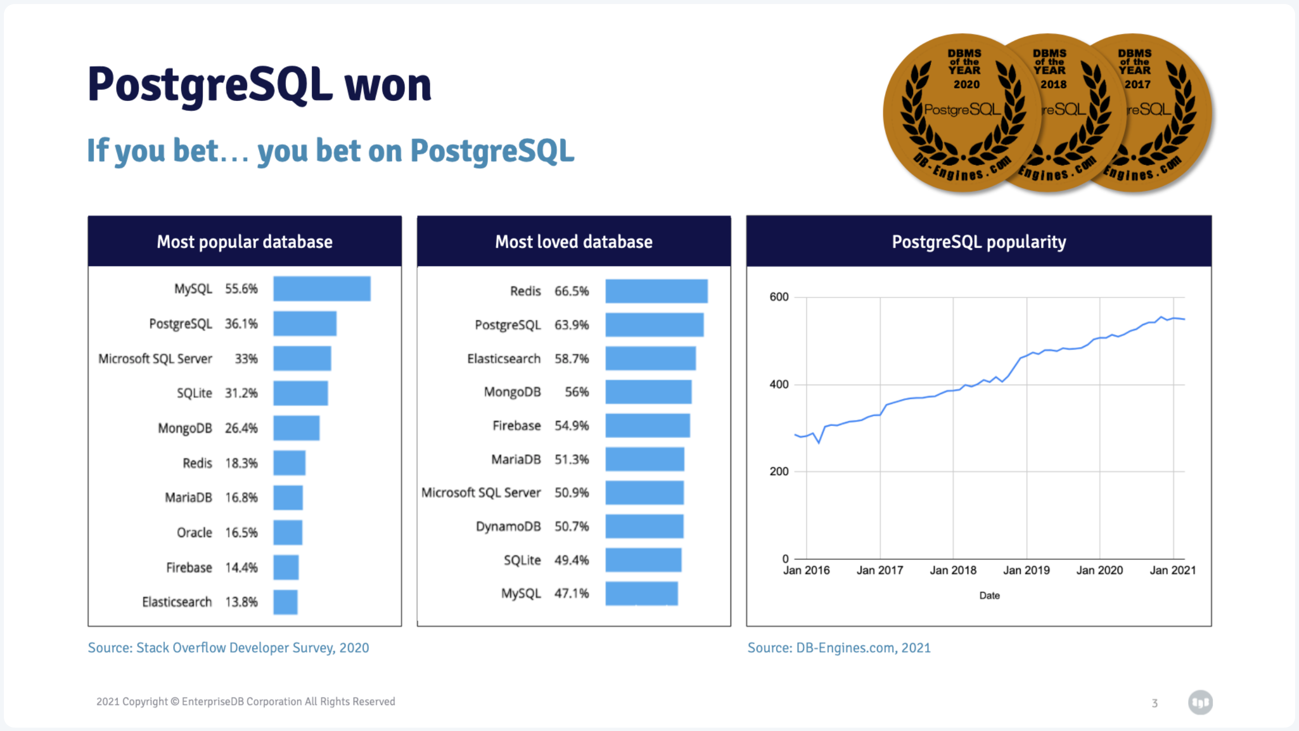


Contents

1 Introduction: How and Why PostgreSQL Works

PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. It has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions.

PostgreSQL has become the open source relational database of choice for many people and organizations, and was recently named **Database Management System of the Year 2020 by DB-Engines**. According to DB-Engines, all of the other major relational databases—MySQL, SQLServer, Oracle, and DB2—have been in slow but steady decline over many years. And yet, PostgreSQL continues to grow. **DB-Engines data shows** that PostgreSQL is growing significantly faster than would-be rivals MongoDB and Redis.



PostgreSQL won

If you bet… you bet on PostgreSQL

Most popular database	
MySQL	55.6%
PostgreSQL	36.1%
Microsoft SQL Server	33%
SQLite	31.2%
MongoDB	26.4%
Redis	18.3%
MariaDB	16.8%
Oracle	16.5%
Firebase	14.4%
Elasticsearch	13.8%

Most loved database	
Redis	66.5%
PostgreSQL	63.9%
Elasticsearch	58.7%
MongoDB	56%
Firebase	54.9%
MariaDB	51.3%
Microsoft SQL Server	50.9%
DynamoDB	50.7%
SQLite	49.4%
MySQL	47.1%

Source: Stack Overflow Developer Survey, 2020

Source: DB-Engines.com, 2021

3

1.1 Standout features

A quick way to describe PostgreSQL is that it is the equivalent in the database area of what Linux represents in the operating system space. **The current latest major version of PostgreSQL is version 13, which ships out of the box:**

- Native streaming replication, both physical and logical
- Parallel queries for vertical scalability
- Declarative partitioning for horizontal table partitioning
- Extensibility, with extensions like PostGIS for geographical databases
- JSON support, which enables the multi-model hybrid database to store both structured/relational and unstructured data and query them via standard SQL
- Continuous hot backup and point in time recovery

In terms of architecture, PostgreSQL natively supports the primary/standby architecture, with optional and multiple replicas. The technology behind it is very robust and it is the evolution of crash recovery and point in time recovery. Replication was first introduced in PostgreSQL 8.2 about 15 years ago through WAL shipping and warm standby, and later improved in PostgreSQL 9.0 through WAL streaming and Hot Standby with read only replicas.

Further improvements include cascading replication to replicate from a standby, synchronous replication to enable RPO=0 clusters and backups at transaction level, and logical replication. Considering that streaming replication has been around for more than 10 years, the technology is very stable and robust and guarantees very high results in terms of business continuity, usually measured by recovery point objective (RPO) and recovery time objective (RTO).

1.2 Running PostgreSQL on Kubernetes

From the previous source, we can now explore how Postgres can be installed in Kubernetes. There are primarily two approaches.

The first one is the basic approach: use self-healing capabilities of Kubernetes by having a pod, which is the smallest unit of deployment in Kubernetes, running a Postgres container with no replica. The volume hosting the Postgres data directory is mounted on the pod and it usually resides in a network storage. Kubernetes simply restarts the pod in case of a problem, or moves it on another Kubernetes node.

The main limitation in this case is represented by the storage, which is the single point of failure. Also results are in general not great for business continuity.

The second approach is through an operator, which is an extension of the Kubernetes controller that defines how a complex application works in business continuity contexts. An operator is currently the state of the art in Kubernetes for this purpose. It simulates the work of a human operator, in an automated and programmatic way.

PostgreSQL can be classified as a complex application and as such would benefit from an operator. An operator not only needs to deploy a cluster (which is the first step), but also to properly react after unexpected events. The typical example is that of a failover.

An operator relies on Kubernetes for capabilities like self-healing, scalability, high availability, updates, access, resource control, and so on. It's designed to be fully automated and to support declarative configuration.

This is what we wanted to do. And this is why we created Cloud Native PostgreSQL.

2 Introducing: Cloud Native PostgreSQL

Cloud Native PostgreSQL (CNP) is an operator for Kubernetes and OpenShift environments, distributed by EDB, that implements the primary/standby architecture using native streaming replication. It works with both PostgreSQL and EDB Postgres Advanced and it is designed to deploy and manage your PostgreSQL clusters in production environments.

This does not mean that it is only useful as a production tool; it is also handy while developing applications.

Although Cloud Native PostgreSQL is primarily designed to work with containerized applications that run in the same Kubernetes cluster and that rely on a PostgreSQL database as their backend, you can also use it with applications that are not in a container.

Here, we'll show the use case of any application developer that wants to easily develop, debug and test their software against a PostgreSQL database on their local machine before hitting the staging environment. Think, for example, about testing some applications whose database workload is split between OLTP and OLAP: you want your OLTP traffic to be executed against the cluster primary server and to offload your OLAP traffic on replicas. In such a case, it is far easier to deploy a PostgreSQL cluster using CNP than setting up a PostgreSQL cluster.

Now, we'll walk through how to:

- Set up a Kubernetes sandbox cluster on your local development environment using Kind
- Install the Cloud Native PostgreSQL operator in your sandbox Kubernetes cluster
- Deploy a PostgreSQL cluster in your sandbox cluster
- Connect your application to the PostgreSQL cluster using services and secrets
- Use the "port-forward" command in kubectl to expose the service outside the sandbox cluster for development purposes

2.1 How to create a Kubernetes sandbox cluster

Note: This section is about installing a sandbox Kubernetes cluster on your local machine with Kind. Feel free to skip it if you have already done it or if you already have a Kubernetes cluster at reach.

First of all, if you have not already configured it in your development environment, you will need a Docker installation. The **Install Docker Engine page** on the official **Docker engine website** contains installation instructions for many platforms.

To test if your Docker installation works fine you can use the hello-world image like in the following example:

```
$ docker run --rm hello-world
This message shows that your installation appears to be
working correctly.
```

This means that your Docker engine is working correctly! Now it's time to install Kind.

Kind, standing for "Kubernetes IN Docker", is a great tool to create a Kubernetes cluster in your local environment. The good thing about it is that, despite being lightweight, Kind is using the same executables as a real production one. This installation is still a CNCF-conformant Kubernetes and it is a way to implement infrastructure abstraction in your development process - which is an important DevOps capability.

You can install "kind" using your preferred package manager or by downloading it from the **Releases page** in the project's Github repository. It is important to remember to put the kind executable in a directory included in the PATH environment variable, as this will make invoking it easier. The **Quick Start page** in the Kind documentation has detailed instructions about that.

To test if Kind is installed and working properly you can run the following command:

```
$ kind --version
kind version 0.10.0
```

And now we are ready to create our first Kubernetes cluster! We do this with the following command:

```
$ kind create cluster
Creating cluster “kind” ...
[...]
```

And in just one minute we have our Kubernetes cluster ready for our tests:

```
$ kubectl get nodes
NAME                 STATUS   ROLES                  AGE   VERSION
kind-control-plane   Ready    control-plane,master   62s   v1.20.2
```

2.2 Installing the Cloud Native PostgreSQL operator

You can install Cloud Native PostgreSQL like any other applications in Kubernetes: using a manifest file.
You can install the latest released version of CNP (1.3.0 at the time of writing this article) by running:

```
$ kubectl apply -f \
    https://get.enterprisedb.io/cnp/postgresql-operator-1.3.0.yaml
namespace/postgresql-operator-system created
[...]
```

You can check if everything is working by looking at the status of the pods in the postgresql-operator-system namespace:

```
$ kubectl get pods -n postgresql-operator-system
NAME                                                     READY   STATUS
RESTARTS   AGE
postgresql-operator-controller-manager-5b9c5d8dbd-jnln6   1/1     Running
0          99s
```

The pod is running and everything is ready. You can find more information about installing Cloud Native PostgreSQL on the **Installation page** of the documentation website.

2.3 Deploying a minimal PostgreSQL cluster

While Cloud Native PostgreSQL is closed-source software, you are still granted an implicit evaluation license that lasts for 30 days. This does not mean that after 30 days your data is lost! It only means that after 30 days the operator will stop reconciling your cluster specification with the status: self-healing features and configuration changes will no longer work.

This can be enough to quickly test a PostgreSQL cluster while it is not certainly enough for your production environment. More information about production plans and subscriptions can be found on the **License Keys page** in the documentation.

Enough with talking, we can start deploying a cluster:

```
$ kubectl apply -f \    https://docs.enterprisedb.io/cloud-native-post-
gresql/1.3.0/samples/cluster-example.yaml
cluster.postgresql.k8s.enterprisedb.io/cluster-example created
```

The cluster definition referenced from the previous command is really simple:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  storage:
    size: 1Gi
```

All we require is a cluster with 3 replicas, each with 1 gigabyte of space. Obviously, the Cluster CRD is more complex than this, and the defaulting webhook will complete the specification for us.

When everything will be ready you will find one Pod per instance (the operation will take a minute or so).
Just like this:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
cluster-example-1   1/1     Running   0          2m51s
cluster-example-2   1/1     Running   0          2m30s
cluster-example-3   1/1     Running   0          2m11s
```

2.4 Services for Cloud Native PostgreSQL

In a traditional VM/physical environment, when accessing a PostgreSQL database from an application you normally need an IP address or a host name. Kubernetes abstracts this and provides a kind of object for clients to connect to a given service. Surprise, surprise ... that resource in Kubernetes is called “Service”. **Cloud Native PostgreSQL automatically provides and 4 services for each cluster:**

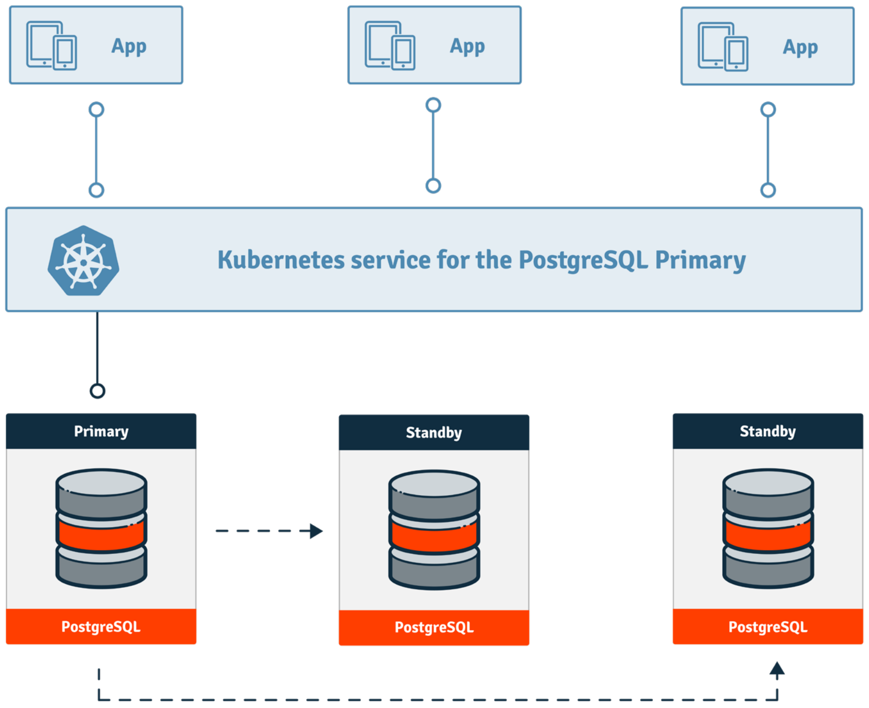
```
$ kubectl get services -o name
service/cluster-sample-any
service/cluster-sample-r
service/cluster-sample-ro
service/cluster-sample-rw
```

If you need to work with the primary server, you can just use the cluster-example-rw service, which will handle read&write traffic.

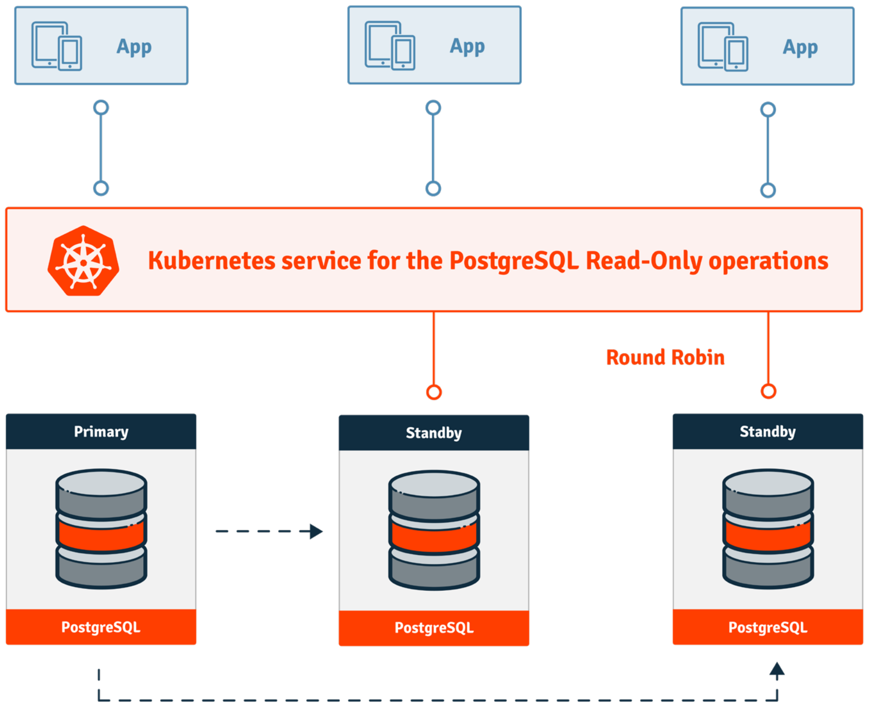
Instead, if you only need to read from the database you can just offload traffic to `cluster-example-ro` to use replicas or to `cluster-example-r` to use the replicas and the primary server too. Kubernetes will take care to keep the services synchronized with the actual PostgreSQL cluster status, following unexpected events like failovers or planned ones such as switchovers or updates.

The following diagram shows what happens when an application uses the cluster-example-rw service. The application writes data to the cluster primary server, which replicates it to the secondary servers:

The following diagram shows what happens when an application uses the cluster-example-rw service. **The application writes data to the cluster primary server, which replicates it to the secondary servers:**



For read-only traffic, queries can be executed against any of the replicas. **The following diagram shows what happens when an application uses the cluster-example-ro service:**



2.5 Credentials (secrets)

What about your credentials? Just look into the generated secrets:

```
$ kubectl get secrets -o name
secret/cluster-sample-app
secret/cluster-sample-ca
secret/cluster-sample-replication
secret/cluster-sample-server
secret/cluster-sample-superuser
secret/cluster-sample-token-5b5jm
```

As an application, you usually will not need superuser privileges to access PostgreSQL. A new database named app owned by a user named "app" has already been created for you, and you can access it using the credentials you will find in the cluster-example-app secret.

The next command will dump your credentials (randomly generated), encoded in base64:

```
$ kubectl get secret cluster-example-app -oyaml -o=jsonpath={.data}
{"password":"REDACTED","pgpass":"REDACTED","username":"REDACTED"}
A quick way to grab your password is:
$ kubectl get secret cluster-example-app -oyaml -o=jsonpath={.data.pass-
word}|base64 -d
REDACTED
```

When you deploy your application inside the same Kubernetes cluster, you will not need to do that, since you can directly use that secret inside the Deployment of the stateless application.

2.6 How to use it from your development

While it is certainly possible to just exec psql within your pods to access the actual PostgreSQL instance running inside, it is easier to map the 5432 port corresponding to a certain service to a local port.

You can do this via the following command:

```
$ kubectl port-forward service/cluster-example-rw 5454:5432 &
Forwarding from 127.0.0.1:5454 -> 5432
Forwarding from [::1]:5454 -> 5432
```

Now you can reach the PostgreSQL primary server (we used the cluster-example-rw service) using your local 5454 port. **Just use the password you extracted before in the previous section:**

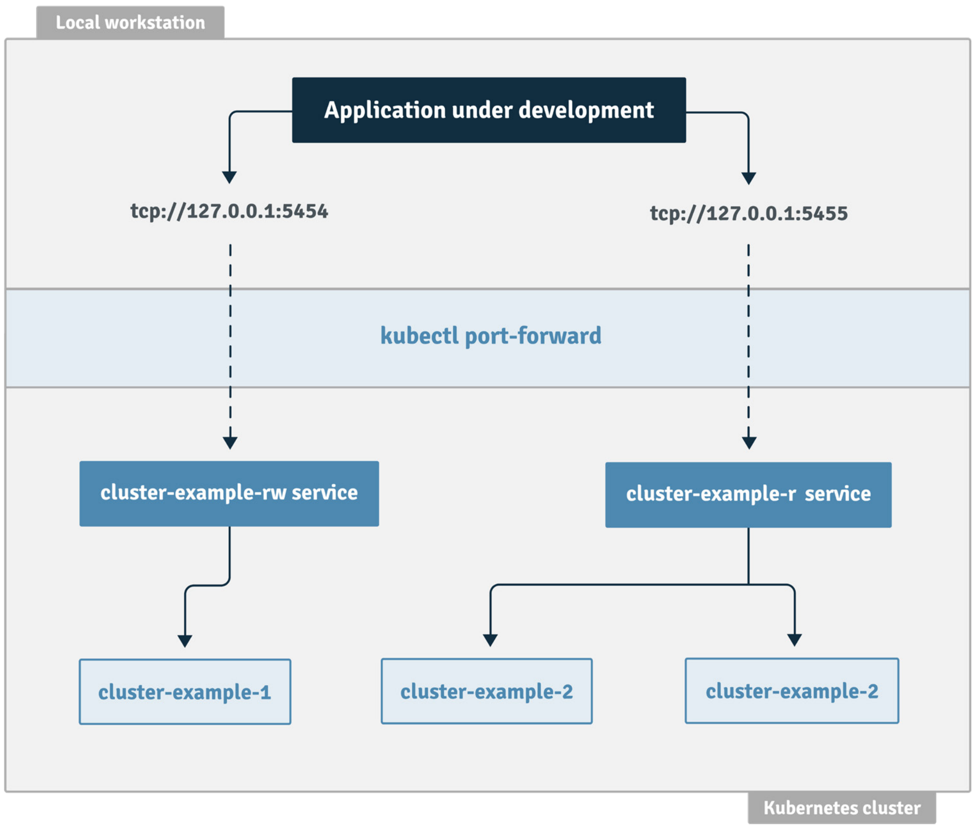
```
$ psql -p 5454 -h 127.0.0.1 -U app app
Password for user app: [...]
app=> \conninfo
SSL connection (protocol: TLSv1.3, [...])
app=> select pg_is_in_recovery();
 f
(1 row)
```

It should not go unnoticed that by default SSL communication is enabled by CNP. **Should you want to have your replicas available as a local port you can just run:**

```
$ kubectl port-forward service/cluster-example-ro 5455:5432 &
And then use port 5455:
$ psql -p 5455 -h 127.0.0.1 -U app app
Password for user app:
app=> select pg_is_in_recovery();
 t
(1 row)
```

As you can see, the PostgreSQL instance is in continuous recovery mode, meaning it is a replica with Hot Standby.

The port forwarding technique works with remote clusters too, and it is surprisingly fast.



3 Conclusion

As you have seen, creating a sandbox Kubernetes environment with a PostgreSQL cluster is very easy, light and quick to set up. **Most importantly, it is self-contained, meaning that it can be easily turned down at the end of your work with:**

```
$ kind delete cluster
```

When used with PostgreSQL, the implicit 30 days usage license is suitable for local development and testing, including automated tests in CI/CD pipelines hosted in Jenkins, Gitlab or Jenkins, to name a few. This is an important step towards abstraction of infrastructure, which reduces variability between development, staging and production environments and increases velocity for your products and software.

To learn more and dive deeper on how to run PostgreSQL in a cloud native environment with Kubernetes, please check out the following blog posts on the EDB blog:

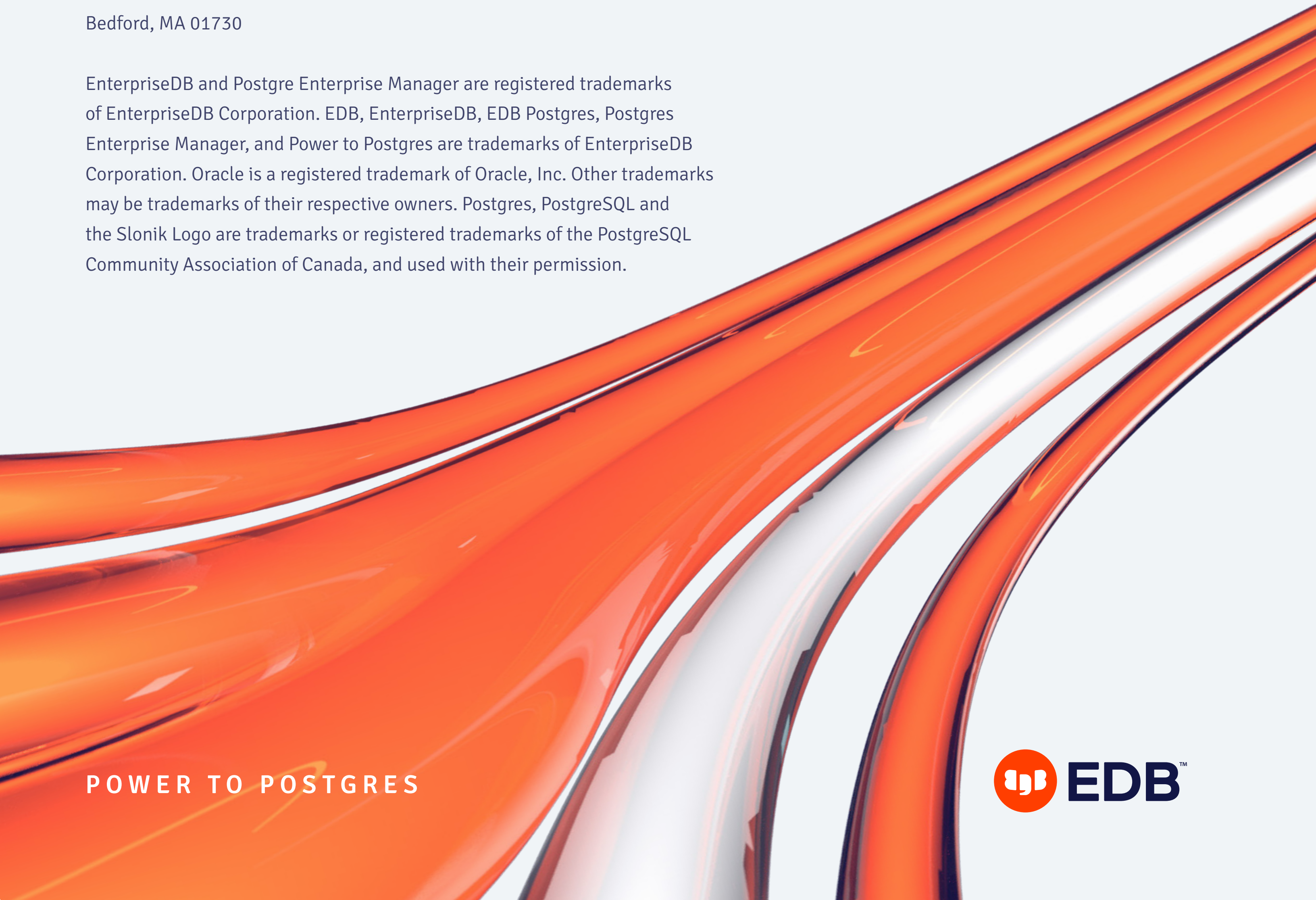
Why EDB Chose Immutable Application Containers
The 4C's Security Model in Kubernetes
Security and Containers in Cloud Native PostgreSQL
For even more, you can visit our **PostgreSQL Experts blog category** and filter for Kubernetes posts.

About EDB

PostgreSQL is increasingly the database of choice for organizations looking to boost innovation and accelerate business. EDB's enterprise-class software extends PostgreSQL, helping our customers get the most out of it both on premises and in the cloud. And our 24/7/365 global support, professional services, and training help our customers control risk, manage costs, and scale efficiently.

With 16 offices worldwide, EDB serves over 4,000 customers, including leading financial services, government, media and communications, and information technology organizations. To learn about PostgreSQL for people, teams, and enterprises, visit EDBpostgres.com.

Running PostgreSQL Database in a Cloud Native Environment with Kubernetes



POWER TO POSTGRES