TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de La Laguna

TECNOLÓGICO NACIONAL DE MÉXICO®

Ingeniería en Sistemas Computacionales

Administración de Bases de Datos

Proyecto Final: PostgreSQL y FreeBSD

#21130598 Viramontes Gutiérrez Edmundo

Docente - J.Carlos Rodríguez U.

Torreón, Coahuila, México

10 de Junio del 2025

Introducción

Este proyecto nace de la asignación de objetivos primarios, estos siendo el uso específico de FreeBSD como sistema operativo y PostgreSQL como DBMS, para la creación de un cluster de servidores, nuestra implementación se realizó en OCI usando en específico la imagen FreeBSD-15.0-CURRENT-amd64-20250424-f2605f67a13e-zfs y la versión 15.13 de PostgreSQL.

Nuestro arquitectura implementada replicación entre servidores maestro - esclavos, balanceo de carga automático, monitoreo en tiempo real y demás objetivos los cuales se explicarán más a fondo en el documento.

Objetivos del proyecto

Como objetivos principales del proyecto estaba el cumplimiento total de la rúbrica de evaluación la cual incluía los puntos:

- Aplicación: Aplicación funcional y bien diseñada que consume el cluster correctamente, con pruebas exhaustivas
- Instalación SO: Instalación correcta de SO en cada servidor, sin errores, con configuraciones óptimas
- Webmin: Uso completo de Webmin para administrar los servidores, con configuraciones avanzadas en cada servidor
- Replicación y Tolerancia a Fallos (FT): Replicación configurada correctamente en 3 o más servidores con tolerancia a fallos (FT), alta disponibilidad y sin errores
- Plan de respaldos: Plan de respaldos implementado, incluyendo respaldos automáticos y estrategias de recuperación detalladas
- Casos de recuperación: Casos de recuperación bien definidos y probados, incluyendo procedimientos para diversos escenarios de fallo
- Monitoreo: Sistema de monitoreo configurado correctamente para todos los servidores, con alertas y análisis de rendimiento en tiempo real
- Auditoría: Auditoría completamente implementada, con registros detallados de todas las actividades en los servidores y bases de datos
- Automatización: Automatización de procesos implementada correctamente, incluyendo respaldos, monitoreo y recuperación
- Balanceador de cargas: Balanceador de cargas implementado correctamente, distribuyendo tráfico de manera eficiente entre los servidores del cluster

Estos objetivos se cumplieron en su totalidad, su implementación se encuentra en el apartado de desarrollo

Desarrollo

Arquitectura del Sistema

Este proyecto implementa un cluster de bases de datos PostgreSQL con alta disponibilidad, diseñado para garantizar que los datos estén siempre disponibles, incluso si uno o más servidores fallan. Utiliza replicación streaming para mantener copias exactas de los datos en múltiples servidores.

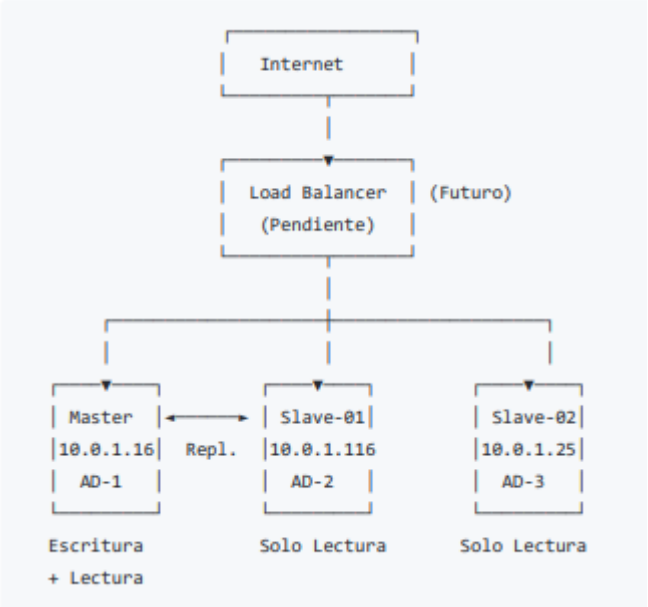


Imagen 1.1 Diagrama de Arquitectura

Como podemos observar en la imagen 1.1 nuestra arquitectura es la conocida Maestra - Servidor en la cual el servidor maestro es el que se encarga de Escritura y Lectura, mientras que los servidores esclavos se encargan solo de lectura. Demostramos la implementación correcta del la creación de los servidores y la instalación de nuestra Imagen de FreeBSD.

Imagen 1.2 Listado de servidores creados

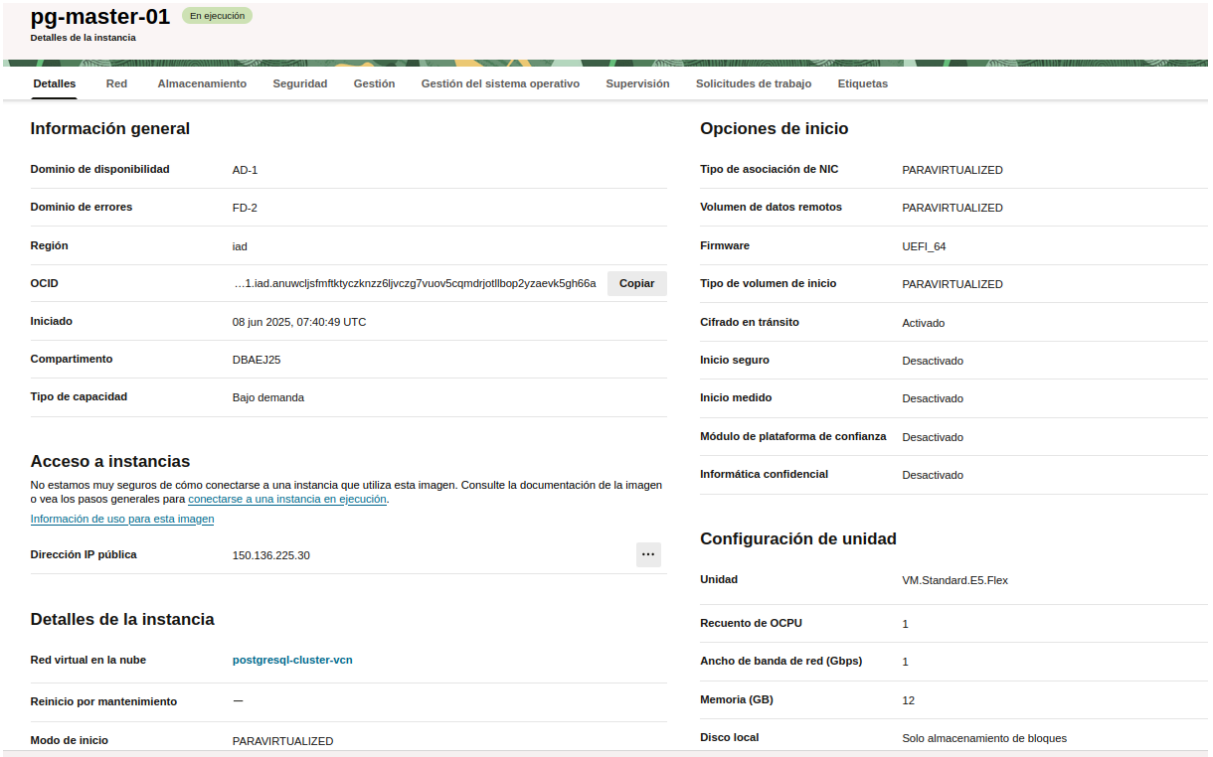


Imagen 1.3 Demostración de Servidos / Instancia con imagen FreeBSD

Cabe recalcar que al momento de la creación de las instancias creamos una nueva VCN la cual lleva el nombre de postgresql-cluster-vcn en la cual se agregaran los puertos necesarios para el desarrollo del proyecto.

Si no te has dado cuenta, YA CUMPLIMOS UN OBJETIVO DE LA RÚBRICA, así es querido lector el objetivo de Instalación SO ya se cumplió, uno menos de la lista hay que seguir.

PostgreSQL

Lo siguiente paso realizado en nuestra configuración de servidores es la instalación y configuración de PostgreSQL, la instalación no es nada fuera de lo normal lo importante aquí es la configuración de nuestros archivos de configuración, me refiero a los cambios realizados en: postgresql.conf y pg_hba.conf.

En postgreSQL hicimos los siguientes cambios:

listen_addresses = ''* = Este comando le cambiamos su parámetro a * pcon el motivo de que sea capaz de permitir conexiones desde cualquier ip, porque recordemos que nuestra arquitectura M-E necesita que los esclavos se conecten remotamente

port = 5432 = Vamos, este es bastante obvio, es el puerto estándar de PostgreSQL

Después pasamos a la configuración de replicación con:

wal_level = replica = este comando define la información de los WAL's, se establece su parámetro como replica porque así se incluye información necesaria para la replicación.

max_wal_senders = 3 = este es el número máximo de precoces que pueden enviar WAL a los esclavos, porque 3? porque querido lector tenemos 2 esclavos y siempre hay que tener espacio de reserva para mantenimiento.

max_replication_slots = 3 = este parámetro define los slots para mantener WAL files para slaves desconectados, lo activamos para que se pierdan cambios si un esclavo se desconecta.

hot_standby = on = este parámetro permite la consulta de solo lecturas en esclavos. porque los esclavos pueden servir consultas mientras replican

Ahora pasemos a la configuración de respaldos, que emoción

archive_mode = on = este parámetro activa el archivado automático de los WAL's para hacer respaldos point-in-time recovery

archive_command = '/usr/local/scripts/backup/backup_incremental.sh %p' = es el comando que ejecuta cada WAL completado, y lu llenado es básicamente la ruta del archivo a archivar

archive_timeout = 300 = este en pocas palabras fuerza archivos WAL cada 5 minutos, esto con el propósito de evitar perder transacciones

Y ahora los cambios en pg_hba.conf fueron los siguientes, recuerden que pg_hba.conf tiene una estructura de llenado específico para quienes se pueden conectar:

host replication replica 10.0.1.0/24 md5 = esta linea permite al usuario 'replica' conectarse desde la red interna, host es la conexión TCP, replication se refiere a base de datos especial para replicación, replica es el nombre del usuario y md5 es la autenticación con contraseña.

host all all 10.0.1.0/24 md5 = esta linea permite Permite conexiones normales desde la red interna, básicamente cualquier usuario a cualquier base de datos

Después de haber estructurado nuestros archivos de configuración, se crea al usuario replica:

CREATE USER replica REPLICATION LOGIN ENCRYPTED PASSWORD 'ulloa';

Ahora pasemos con la configuracion de PostgreSQL en los esclavos:

Webmin

Ahora pasemos a la instalación de webmin en los servidores, este es un proceso bastante sencillo por lo cual se mostrará solamente el resultado. (Usuario: admin, Contraseña: ulloa)

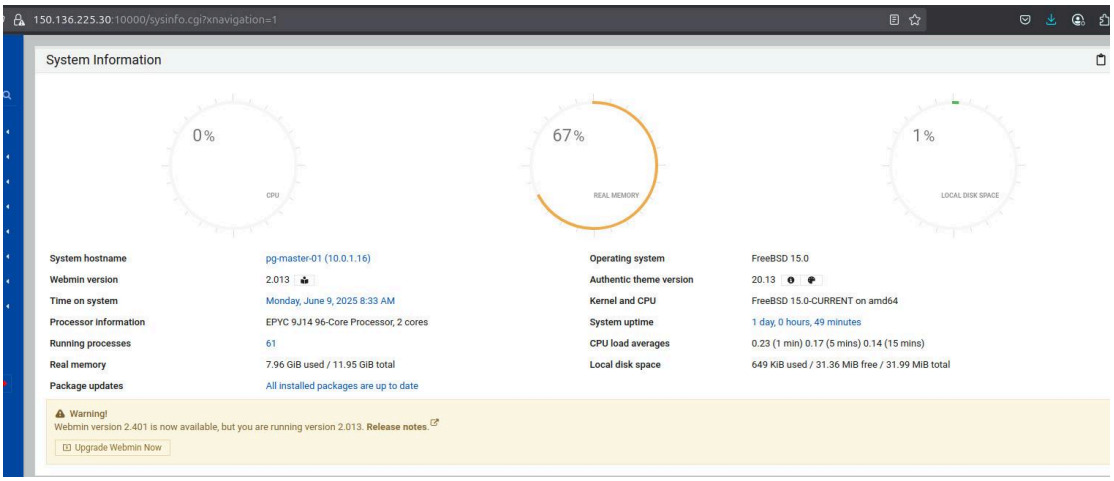


Imagen 2.1 Webmin en servidor maestro

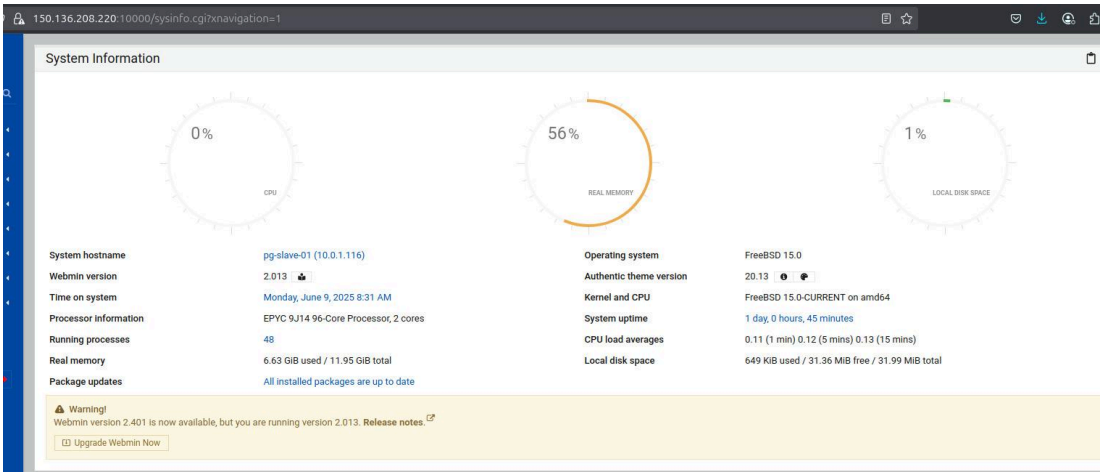


imagen 2.2 Webmin en servidor esclavo 1

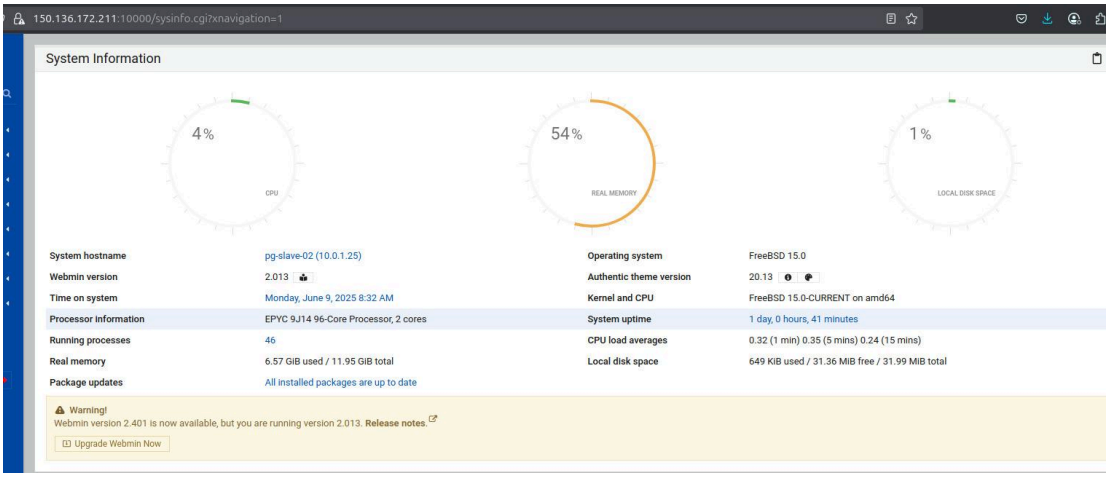


imagen 2.3 Webmin en servidor esclavo 2

Y si lector lo hemos hecho de nuevo, OTRO OBJETIVO CUMPLIDO, este siendo Webmin. Cabe recalcar que para el funcionamiento correcto de lo que acabamos de hablar de tuvo que agregar los puertos a la Security List de nuestra VCN.

Puerto	Servicio	Propósito
22	SSH	Administración remota
5432	PostgreSQL	Base de datos
10000	Webmin	Interfaz web de gestión

Imagen 3.1 Primeros puertos

Replicación

Y también querido lector, si tu lo tienes, sin darnos cuenta hemos cumplido otro objetivo, recuerdas aquellos cambios que hicimos en los archivos de configuración del PostgreSQL, así es, esos cambios son los que nos permiten la replicación, te preguntaras como asi? bueno deja te explico:

Configurando REPLICACIÓN PostgreSQL:

1. Master = Nodo principal (donde se escriben los datos)
2. Slaves = Nodos que copian automáticamente los datos del master

Se configuran los esclavos comos slave

sudo su - postgres -c "echo \"primary_conninfo = 'host=10.0.1.16 port=5432 user=replica password=ulloa' recovery_target_timeline = 'latest'\" > /var/db/postgres/data15/postgresql.auto.conf"

y asi su capacidad de recibir información. Si no me crees toma esta prueba real que realizamos:

En el master:
Crear una tabla de prueba

sudo su - postgres -c "psql -c 'CREATE TABLE test_replication (id serial, mensaje text, fecha timestamp default now());'"

Insertar datos de prueba

sudo su - postgres -c "psql -c 'INSERT INTO test_replication (mensaje) VALUES (\'Prueba desde master\');'"

Ver los datos en el master

sudo su - postgres -c "psql -c 'SELECT * FROM test_replication;'"

y en algún slave:

Verificar si la tabla se replicó

sudo su - postgres -c "psql -c 'SELECT * FROM test_replication;'"

También verificar que las bases de datos están sincronizadas

sudo su - postgres -c "psql -c '\dt'"

Es aquí cuando tenemos que hablar de nuestra de base de datos que utilizamos en nuestro proyecto la cual tiene la siguiente estructura:

```
CREATE TABLE clientes (
cliente_id SERIAL PRIMARY KEY,
nombre VARCHAR(100) NOT NULL,
email VARCHAR(100) UNIQUE NOT NULL
);
CREATE TABLE productos (
producto_id SERIAL PRIMARY KEY,
nombre VARCHAR(100) NOT NULL,
precio NUMERIC(10, 2) NOT NULL CHECK (precio >= 0)
);
CREATE TABLE pedidos (
pedido_id SERIAL PRIMARY KEY,
cliente_id INT NOT NULL REFERENCES clientes(cliente_id) ON DELETE CASCADE,
producto_id INT NOT NULL REFERENCES productos(producto_id),
fecha_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX idx_pedidos_cliente ON pedidos(cliente_id);
CREATE INDEX idx_pedidos_producto ON pedidos(producto_id);
```

Esta es la base de datos que utilizaremos a lo largo del proyecto en la cual a lo largo de la documentación se demostrar que:

Master: Base de datos proyecto_bd con todas las tablas

Slave-01: Datos idénticos replicados automáticamente

Slave-02: Datos idénticos replicados automáticamente

Respaldos

Ahora es momento de demostrar comocumplimos el objetivo de Plan de respaldos, empecemos con lo primero, la creación de directorios para guardar dichos respaldos

```
# Crear directorio para scripts de respaldo
sudo mkdir -p /usr/local/scripts/backup
sudo chown postgres:postgres /usr/local/scripts/backup
# Crear directorio para almacenar respaldos
sudo mkdir -p /var/backups/postgresql
sudo chown postgres:postgres /var/backups/postgresql
# Crear script de respaldo completo
sudo nano /usr/local/scripts/backup/backup_completo.sh
```

Nuestro script de backup_completo.sh (/usr/local/scripts/backup/backup_completo.sh) es aquel que cual nos permite mediante el uso de pg_dumpall (recuerdan eso de nuestra exposición en la clase, espero que si) exportar todas las bases de datos del cluster, en nuestro caso la que mencione anteriormente, osea basicamente este script nos permite generar un archivo SQL que puede recrear todo el cluster.

El script backup_incremental.sh (/usr/local/scripts/backup/backup_incremental.sh) nos permite un respaldo incremental de PostgreSQL (WAL archiving)

Okay ya tenemos nuestros scripts pero y luego esos como se usan o como? bueno querido lector no te tienes que preocupar de nada, porque utilizamos CRON para automatizar y no tener que hacerlo manualmente, que bello no?

Primero configuramos CRON para el usuario postgres y agregamos:

```
# Respaldo completo diario a las 2:00 AM
0 2 * * * /usr/local/scripts/backup/backup_completo.sh

# Respaldo de la base del proyecto cada 6 horas
0 */6 * * * /usr/local/bin/pg_dump -d proyecto_bd | gzip > /var/backups/postgresql/proyecto_bd_$(date
+\%Y\%m\%d_\%H\%M).sql.gz

# Limpiar logs antiguos semanalmente
0 3 * * 0 find /var/backups/postgresql -name "*.log" -mtime +14 -delete
```

WOWOWOW, ya cumplimos con otro objetivo Sistema de Respaldos, ya llevamos cuatro de diez, vamos muy bien; Y si te das cuenta ya estamos casi completando otro objetivo, esté siendo automatización, pero no podemos darlo como completado todavía.

Monitoreo

Por ahora avanzaremos al objetivo de monitoreo aquí tuvimos bastantes problemas ya que intentamos de manera inicial una implementación mediante Prometheus + Grafana pero descubrimos después de muchos errores y horas invertidas que la versión de OS (FreeBSD) que estábamos usando era incompatible con aplicaciones GO y sorpresa Prometheus y Grafana son aplicaciones GO, por lo cual tuvimos que hacer una implementación totalmente diferente y mucho más complicada, esta siendo la creación de scripts personalizados para el monitoreo, si si lo se que loco, pero te voy diciendo que si nos salio.

Primero que nada creamos una carpeta en donde guardar nuestros archivos

sudo mkdir -p /usr/local/scripts/monitoring

Y empezamos la creacion de nuestros scripts, collect_metrics.sh (/usr/local/scripts/monitoring/generate_dashboard.sh) el cual nos permite generar un HTML dinámico con métricas recolectadas, diferencia visualmente entre master y slaves, y muestra el estado online/offline de cada servidor; Despues creamos generate_dashboard.sh (/usr/local/scripts/monitoring/collect_metrics.sh) el cual recolecta métricas de slaves vía SSH.

Después de la creación de los scripts hay que utilizarlos ¿no? bueno nosotros los utilizamos mediante CRON

```
# Recolección de métricas cada 2 minutos (todos los servidores)
*/2 * * * * /usr/local/scripts/monitoring/collect_metrics.sh
# Recolección de slaves cada 3 minutos (solo master)
*/3 * * * * /usr/local/scripts/monitoring/collect_from_slaves.sh
```

Servidor	Métricas Recolectadas
Master	CPU, Memoria, Disco, Conexiones PG, DB Size, **Slaves conectados**
Slaves	CPU, Memoria, Disco, Conexiones PG, DB Size, **Estado replicación**

Ya con nuestros scripts, hay que tener un lugar donde mostrar la información, en nuestro caso creamos un Dashboard de monitoreo (http://150.136.225.30:8080) el cual nos muestre la información que recolectamos de las métricas, este Dashboard paso por muchas iteraciones de diseño esto debido a bugs o una incorrecta implementación, pero bueno al final nuestro dashboard se ve de la siguiente manera:

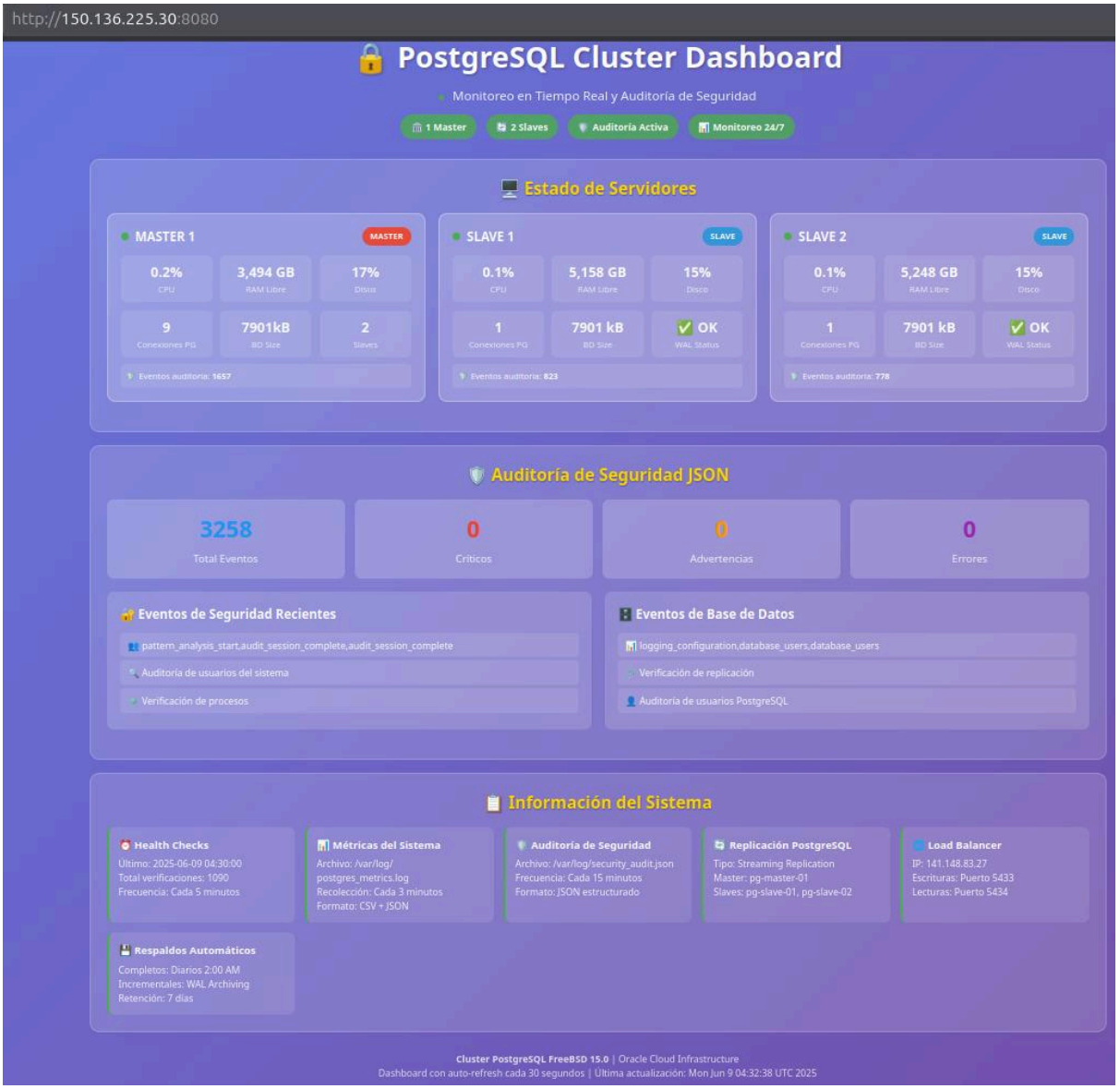


Imagen 4.1 Dashboard final (ignoren la parte de auditoría, eso viene después)

Cabe recalcar que se utilizó nginx, esto con el motivo de tener un web server el cual nos permite tener nuestro Dashboard, procesar archivo PHP, etc, eso se explicará después.

Algunos de los problemas que nos encontramos fue que con los scripts que teníamos se recopila la información sí, pero no se mostraba asi que creamos otro script con su respectiva automatización con CRON, este siendo update_metrics_only.sh (/usr/local/scripts/monitoring/update_metrics_only.sh) el cual nos permite Recolecta métricas frescas del cluster, verifica replicación, corrige estados 'N/A', actualiza HTML específico, preserva personalizaciones y maneja errores robustamente.

#Comando CRON para ejecutar el script cada minuto

* * * * * /usr/local/scripts/monitoring/update_metrics_only.sh >> /var/log/dashboard_updates.log 2>&1

Métrica	Estado
Actualización dashboard	**Automática**
Frecuencia de datos	**Cada 30 segundos**
Estado WAL Slave1	**OK (correcto)**
Estado WAL Slave2	**OK**
Intervención requerida	**No**

Entonces podemos decir que hemos cumplido el objetivo de monitoreo? Yo digo que si, en especial con mas explicacion dentro del desarrollo de otros objetivos como auditoría

Automatización

Este objetivo puede ser difícil de asimilar ya que su simple existencia esta ligada con otros objetivos pero hagamos un recuento de los procesos que se han automatizado, primero recordemos postgresql.conf en el cual se establece:

```
archive_mode = on
archive_command = '/usr/local/scripts/backup/backup_incremental.sh %p'
archive_timeout = 300 # Cada 5 minutos
```

Después pasemos a CRON:

```
# Respaldo completo diario a las 2:00 AM
0 2 * * * /usr/local/scripts/backup/backup_completo.sh
# Respaldo del proyecto cada 6 horas
0 */6 * * * /usr/local/bin/pg_dump -d proyecto_bd | gzip > /var/db/postgre
# Recolección de métricas cada 2 minutos
*/2 * * * * /usr/local/scripts/monitoring/collect_metrics.sh
# Recolección de slaves cada 3 minutos
*/3 * * * * /usr/local/scripts/monitoring/collect_from_slaves.sh
# Limpieza semanal de logs
0 3 * * 0 find /var/log -name "*.log" -mtime +14 -delete
# Solo recolección de métricas locales */2 * * * *
/usr/local/scripts/monitoring/collect_metrics.sh
* * * * * /usr/local/scripts/monitoring/update_metrics_only.sh >> /var/log/dashboard_updates.log 2>&1
# Crontab entry en cada servidor:
*/5 * * * * /usr/local/scripts/health_checks/service_monitor.sh
```

Esos son las automatizaciones iniciales, después de esas primeras también implementamos más como la creacion del script service_monitor.sh (existe en los tres servidores) el cual monitorea PostgreSQL, nginx y Webmin automáticamente, reinicia servicios caídos, registra eventos en logs y notifica problemas críticos.

Ejemplo de resultados:

```
2025-06-08 20:40:00,[OK],pg-master-01,postgresql,PostgreSQL accepting connections 2025-06-08
20:40:00,[OK],pg-master-01,postgresql,Database proyecto_bd accessible 2025-06-08
```

20:40:00,[OK],pg-master-01,nginx,nginx listening on port 8080 2025-06-08
20:40:00,[OK],pg-master-01,nginx,nginx responding to HTTP requests 2025-06-08
20:40:00,[OK],pg-master-01,webmin,webmin listening on port 10000 2025-06-08
20:40:00,[OK],pg-master-01,system,Disk usage normal: 17% 2025-06-08
20:40:00,[OK],pg-master-01,system,Memory available: 4695944MB free 2025-06-08
20:40:00,[INFO],pg-master-01,health_check,All health checks passed

Load Balancer

Es un proyecto muy grande por lo que saltos entre diferentes objetivos y desarrollos serán comunes, así que hablemos del Load Balancer o Balanceador de Carga, esté lo implementamos con la herramienta de OCI el cual nos permite de manera fácil y eficiente implementar nuestro balanceador de carga, te preguntaras como es que en un estructura Maestro - Esclavo se utiliza el balanceo de carga, bueno primero no hay que caer en la trampa de confundir replicación y balanceo, en nuestro caso la replicación es unidireccional ya que el maestro es el que manda a los esclavos y no viceversa, pero en nuestro caso el balanceo se da la siguiente manera:

Cliente ↔ Master: Para writes (INSERT/UPDATE/DELETE)
Cliente ↔ Slaves: Para reads (SELECT)

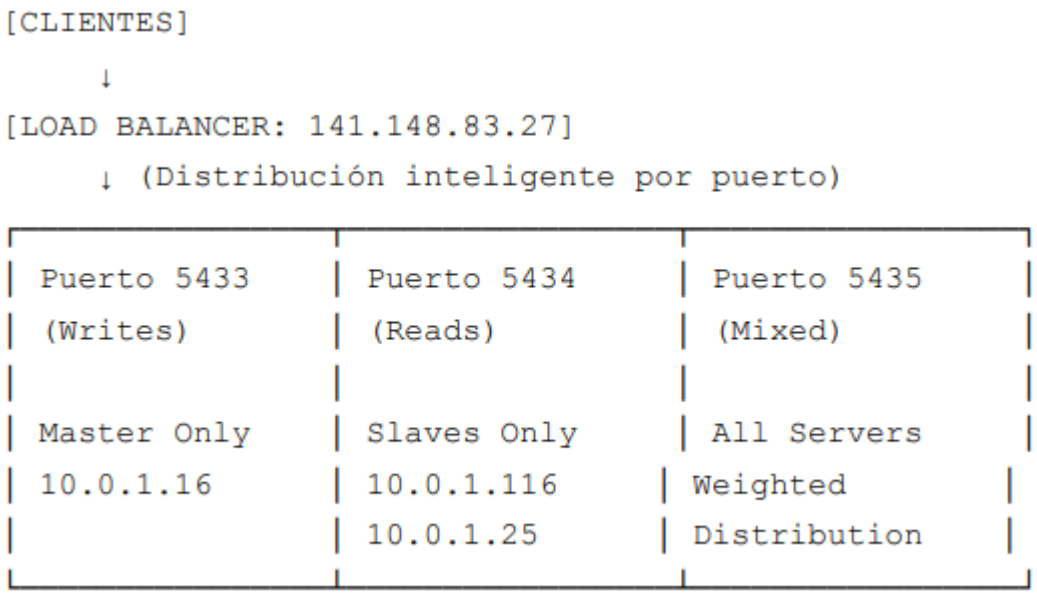
Ahora hablemos de nuestra arquitectura del balanceador, es importante saber primero que el Load Balancer de OCI solo se maneja a nivel de red (Layer 4) - solo ve TCP packets, por lo cual no diferencia automáticamente entre queries de escritura vs lectura, por lo cual tomando dicha consideración en cuenta optamos por configurar nuestro Load Balancer con tres configuraciones:

Backend Set #1: "postgres-writes"
Servers: Solo 10.0.1.16:5432 (Master)
Health Check: TCP port 5432

Backend Set #2: "postgres-reads"
Servers: 10.0.1.116:5432 + 10.0.1.25:5432 (Slaves)
Health Check: TCP port 5432

Backend Set #3: "postgres-mixed"
Servers: 10.0.1.16:5432 + 10.0.1.116:5432 + 10.0.1.25:5432
Health Check: TCP port 5432

Listeners:
Listener 1: Puerto 5433 → Backend Set "postgres-writes"
Listener 2: Puerto 5434 → Backend Set "postgres-reads"
Listener 3: Puerto 5435 → Backend Set "postgres-mixed"



El primer backend "postgres-writes" tiene el propósito de las operaciones de escritura exclusivamente al Master, el segundo backend "postgres-reads" tiene el propósito de las operaciones de lectura distribuidas entre Slaves y el backend "postgres-mixed" tiene el propósito de la distribución balanceada con preferencia al Master

Parámetro	Valor
Nombre	postgres-writes
Política	Asignación en rueda ponderada
Health Check	TCP puerto 5432
Servidores	Solo pg-master-01 (10.0.1.16:5432)
Peso	1

Imagen 6.1 postgres-writes

Parámetro	Valor
Nombre	postgres-reads
Política	Asignación en rueda ponderada
Health Check	TCP puerto 5432
Servidores	pg-slave-01 (10.0.1.116:5432) pg-slave-02 (10.0.1.25:5432)
Peso	1 cada uno

Imagen 6.2 postgres-reads

Parámetro	Valor
Nombre	postgres-mixed
Política	Asignación en rueda ponderada
Health Check	TCP puerto 5432

Parámetro	Valor
Servidores	pg-master-01 (10.0.1.16:5432) **peso 2** pg-slave-01 (10.0.1.116:5432) **peso 1** pg-slave-02 (10.0.1.25:5432) **peso 1**

Imagen 6.3 postgres-mixed

Después de tener la configuración correcta de nuestro Load Balancer (BackEnds y Listeners) hay que agregar los puertos a nuestra Security List para su correcto uso.

Quiero recalcar algo importante acerca de lo que está realizando nuestro balanceador su función es el balanceo de conexiones PostgreSQL (base de datos) mediante el protocolo TCP en puertos 5433, 5434, 5435, básicamente distribuir queries SQL entre Master/Slaves no es para aplicaciones web HTTP/HTTPS.

Flujo de la Arquitectura del Load Balancer

[Usuario en navegador]
↓ HTTP
[NGINX Puerto 8081] ← Aplicación Web
↓
[NGINX Puerto 8082] ← Backend PHP APIs
↓ SQL/TCP
[Load Balancer 141.148.83.27] ← Distribución PostgreSQL
↓
[Master/Slaves PostgreSQL]

Recordemos que:

Cliente Web → NGINX (150.136.225.30:8081) → Frontend
Frontend → NGINX (150.136.225.30:8082) → PHP Backend
PHP Backend → Load Balancer (141.148.83.27:5433/5434) → PostgreSQL

Auditoría

Si analizamos extensivamente el proyecto podríamos decir que ya tenemos avanzado auditoría por que en postgresql.conf tenemos lo siguiente:

```
# Logging básico
logging_collector = on
log_destination = 'stderr'
log_directory = 'log'
```

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
# Eventos auditados
log_connections = on
log_disconnections = on
log_statement = 'mod' # DDL y DML
log_min_duration_statement = 1000 # Queries >1 segundo
# Formato con contexto
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
```

Pero si somos reales, no es en verdad auditoría ver un log, asi que hay que cumpli bien el objetivo, se implementó un Sistema de Logs JSON Estructurados con la siguiente estructura

```
{
 "timestamp": "2025-06-08T22:03:00-0600",
 "audit_session_id": "audit_1717890180_12345",
 "hostname": "pg-master-01",
 "server_role": "master",
 "event_type": "SECURITY",
 "severity": "INFO",
 "category": "users",
 "event_name": "audit_start",
 "message": "Iniciando auditoría de usuarios del sistema",
 "script_version": "1.0",
 "details": {}
}
```

Campos del Log JSON:

- timestamp: Fecha/hora en formato ISO 8601
- audit_session_id: ID único de la sesión de auditoría
- hostname: Nombre del servidor (pg-master-01, pg-slave-01, pg-slave-02)
- server_role: Rol del servidor (master/slave)
- event_type: Tipo de evento (SECURITY, DATABASE, SYSTEM, NETWORK)
- severity: Nivel de severidad (INFO, WARNING, ERROR, CRITICAL)
- category: Categoría específica (users, processes, files, network)
- event_name: Nombre del evento específico
- message: Descripción detallada del evento
- script_version: Versión del script de auditoría
- details: Objeto JSON con información adicional

Después de implemento un sistema de Auditoría de Seguridad del Sistema el cual abarca multiples casos:

A) Auditoría de Usuarios del Sistema
Verificaciones Implementadas:

- Usuarios con UID 0: Detección de múltiples usuarios root
- Usuarios sin contraseña: Identificación de cuentas inseguras
- Usuarios con shell access: Listado de usuarios con acceso de terminal
- Últimos logins exitosos: Histórico de accesos recientes

Ejemplo de Evento Generado:
{

"event_type": "SECURITY",
"severity": "WARNING",
"category": "users",
"event_name": "multiple_root_users",
"message": "Múltiples usuarios con UID 0 detectados",
"details": {
"root_users": ["root", "toor"],
"count": 2
}
}

B) Auditoría de Procesos
Verificaciones Implementadas:

- Procesos ejecutándose como root: Top 10 procesos con privilegios
- Alto uso de CPU: Detección de procesos con >10% CPU
- Puertos en escucha: Mapeo de servicios activos
- Procesos PostgreSQL: Conteo y estado de procesos de BD

Ejemplo de Evento Generado:
{
"event_type": "SECURITY",
"severity": "WARNING",
"category": "processes",
"event_name": "high_cpu_usage",
"message": "Procesos con alto uso de CPU detectados",
"details": {
"high_cpu_processes": "postgres:15.2;nginx:12.5;perl:8.3"
}
}

C) Auditoría de Archivos Críticos
Archivos Monitoreados:

- /etc/passwd - Usuarios del sistema
- /etc/master.passwd - Contraseñas encriptadas
- /var/db/postgres/data15/postgresql.conf - Configuración PostgreSQL
- /var/db/postgres/data15/pg_hba.conf - Autenticación PostgreSQL
- /usr/local/etc/nginx/nginx.conf - Configuración nginx
- Verificaciones por Archivo:
- Permisos: Usuario:Grupo:Permisos
- Checksum SHA256: Integridad del archivo
- Última modificación: Timestamp de cambios
- Propietario correcto: Validación de ownership

Ejemplo de Evento Generado:
{
"event_type": "SECURITY",
"severity": "INFO",
"category": "files",

"event_name": "critical_file_status",
"message": "Estado de archivo crítico: /etc/passwd",
"details": {
"file_path": "/etc/passwd",
"permissions": "-rw-r--r--:root:wheel",
"checksum": "a1b2c3d4e5f6...",
"last_modified": "Jun 8 15:30:22 2025"
}
}

D) Auditoría de Conexiones de Red
Verificaciones Implementadas:

- Conexiones SSH activas: Usuarios conectados remotamente
- Conexiones PostgreSQL: Conteo de sesiones de BD activas
- Puertos abiertos inesperados: Detección de servicios no autorizados
- Conectividad entre nodos: Verificación de comunicación cluster

Puertos Esperados:

- 22: SSH
- 5432: PostgreSQL
- Page 5 of 17
- 8080: nginx (solo master)
- 10000: webmin

E) Auditoría Específica PostgreSQL
Verificaciones de Base de Datos:

- Configuración SSL: Estado de encriptación
- Logging de statements: Configuración de auditoría
- Usuarios de PostgreSQL: Roles y privilegios
- Métodos de autenticación: Configuración pg_hba.conf

F) Análisis de Patrones Sospechosos
Detección Automática:

- Intentos de login fallidos: >5 intentos en auth.log
- Uso de disco crítico: >80% de ocupación
- Procesos sospechosos: netcat, telnet, ftp activos

Aquí es importante recalcar que se adecuo el Dashboard de monitoreo para también soportar auditoria, en el sentido que agrego Estadísticas del Cluster, Eventos Dinámicos Mostrados :

Estadísticas del Cluster (4 Tarjetas Principales):

Total Eventos (Azul #2196F3):

- Número Grande: X (suma de eventos de los 3 servidores)
- Descripción: "Total Eventos"
- Significado: Contador acumulativo de todos los eventos JSON de auditoría

Eventos Críticos (Rojo #f44336):

- Número: 0 (ideal para seguridad)
- Descripción: "Críticos"
- Significado: Eventos que requieren atención inmediata (fallos de seguridad, intrusiones)

Advertencias (Naranja #FF9800):
- Número: 0 (estado óptimo)
- Descripción: "Advertencias"
- Significado: Situaciones que requieren monitoreo (uso alto de recursos, configuraciones
- subóptimas)

Errores (Morado #9C27B0):
- Número: 0 (sistema estable)
- Descripción: "Errores"
- Significado: Errores detectados que necesitan corrección (fallos de servicios, conectividad)

Eventos Recientes (2 Paneles Lado a Lado):

Panel Izquierdo - Eventos de Seguridad:
- Título: " Eventos de Seguridad Recientes"
- Eventos Dinámicos Mostrados:
- [Eventos específicos]: audit_start, users_check, files_check (extraídos de logs reales)
- Auditoría de usuarios del sistema: Verificación de cuentas y privilegios
- Verificación de procesos: Análisis de procesos en ejecución

Panel Derecho - Eventos de Base de Datos:
- Título: " Eventos de Base de Datos"
- Eventos Dinámicos Mostrados:
- [Eventos específicos]: postgresql_check, replication_status, connections_audit
- Verificación de replicación: Estado de streaming replication
- Auditoría de usuarios PostgreSQL: Roles y permisos de BD

Entonces ya podemos decir que cumplimos auditoría WUUU.

Aplicación Web

Vamos con el mero mole, la representación gráfica de la mayoría de las cosas que hemos estado hablando, la rúbrica nos pide Aplicación funcional y bien diseñada que consume el cluster correctamente, con pruebas exhaustivas. En nuestro caso una implementación completa que consume el cluster PostgreSQL a través del Load Balancer Oracle Cloud, demostrando el funcionamiento end-to-end del sistema de alta disponibilidad.

Okay, nosotros lo realizamos de la siguiente manera:

1. Frontend Web moderno en HTML/CSS/JavaScript
2. Backend PHP con APIs REST funcionales
3. Integración completa con Load Balancer Oracle Cloud
4. CRUD funcional conectado al cluster PostgreSQL
5. Testing exhaustivo del sistema completo

Diagrama de Componentes

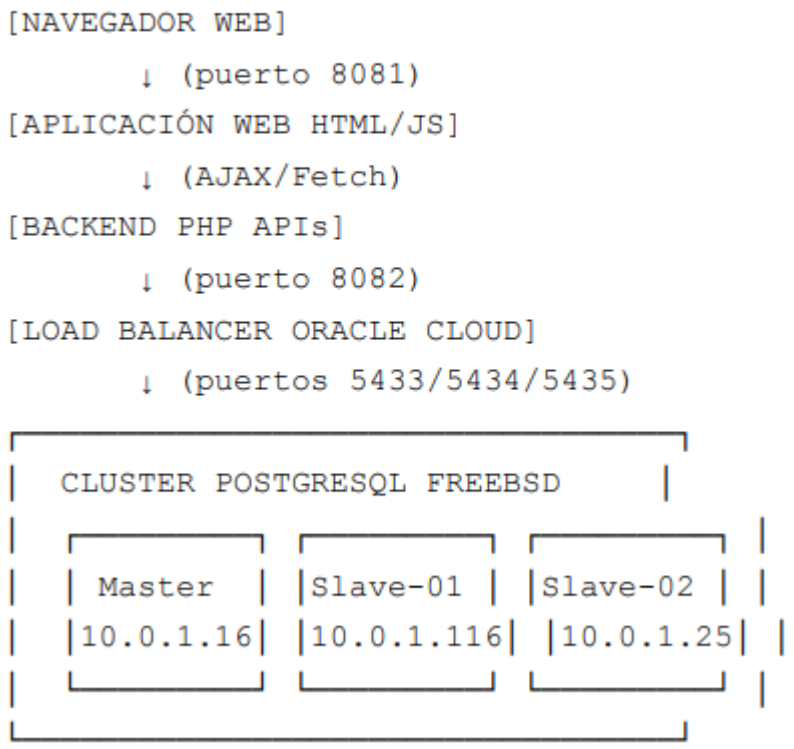


Imagen 7.1 Diagrama de Componentes

Después del planteamiento inicial de componentes lo primero que se realizó fue la configuración de nginx (/usr/local/etc/nginx/nginx.conf), para adecuar los nuevos puertos, y después se instaló PHP PostgreSQL para el manejo del BackEnd; Nuestra estructura de directorios se debe de ver así:

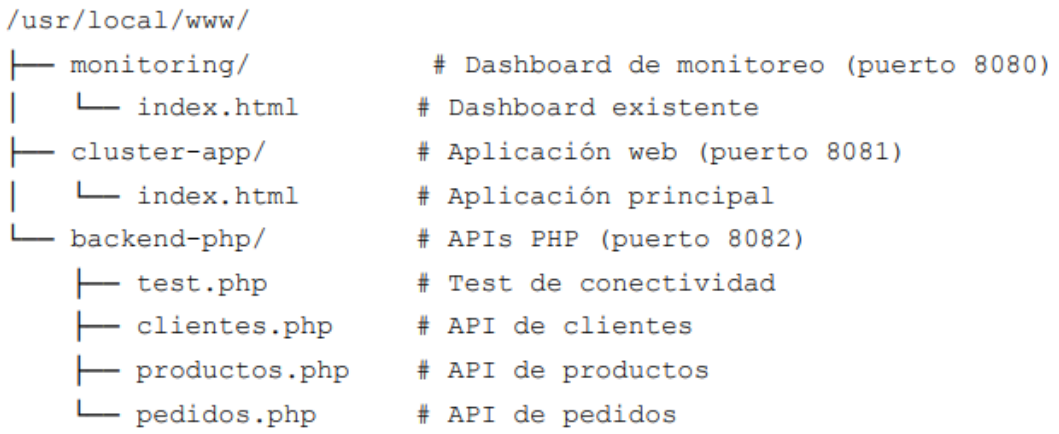


Imagen 7.2 Estructura de directorios

Ahora que ya tenemos nuestra configuración inicial lo que hicimos fue el Backend PHP, la parte en la que más fallos por errores tontos hubo, como primer paso se realizó un test.php con el fin de probar funcionalidades básicas sin arruinar nuestra estructura ni archivos ya creados, al realizar las pruebas básicas se desarrollaron los archivos API clientes.php, productos.php y pedidos.php, estos en pocas palabras manejan la funcionalidad backend de CRUD de sus respectivas tablas.

Notas / Características de estos archivos:

- Separación read/write: Lecturas al puerto 5434 (slaves), escrituras al 5433 (master)
- Validación de datos: Verificación de email único antes de insertar
- Prepared statements: Protección contra SQL injection
- RETURNING clause: Obtener ID del registro insertado
- Cascading deletes: Eliminación automática de pedidos relacionados
- JSON input/output: Comunicación estándar con el frontend

Ya con el backend completado se desarrolló el Frontend, el GUI se realizó con HTML5, CSS3 y JavaScript ES6+ (se que te lo preguntaste, si, si es responsiva la aplicación) y la comunicación con el Backend mediante un Fetch API.

El código del Frontend es muy largo por lo cual si queremos ver el código en lo mejor sería ser verlo mediante webmin aquí solo mencionaré algunos fragmentos importantes:

```
const CONFIG = {
 backendUrl: 'http://150.136.225.30:8082',
 endpoints: {
 test: '/test.php',
 clientes: '/clientes.php',
 productos: '/productos.php',
 pedidos: '/pedidos.php'
 }
};
```

```
// Función genérica para llamadas API con manejo de errores
async function apiCall(endpoint, method = 'GET', data = null) {
 try {
 const url = `${CONFIG.backendUrl}${CONFIG.endpoints[endpoint]}`;
 const options = {
 method: method,
 headers: {
 'Content-Type': 'application/json',
 }
 };
 if (data && method === 'POST') {
 options.body = JSON.stringify(data);
 }
 const response = await fetch(url, options);

 if (!response.ok) {
 throw new Error(`HTTP error! status: ${response.status}`);
 }

 const result = await response.json();
 return result;
 } catch (error) {
 console.error('API call failed:', error);
 showAlert(`Error de conexión: ${error.message}`, 'error');
 updateApiStatus(false, `Error: ${error.message}`);
 throw error;
 }
}

// Test de conectividad con resultados reales del backend
async function testClusterConnection() {
 try {
 showLoading(true);
 const result = await apiCall('test');

 const testResults = document.getElementById('test-results');
 if (result.status === 'success') {
 const resultHtml = `
 <div class="alert alert-success">
 <strong> Test de Conexión Backend Exitoso</strong><
 <strong>Load Balancer:</strong> ${result.load_balance
 <strong>Write Test:</strong> Servidor ${result.write_
 <strong>Read Test:</strong> Servidor ${result.read_te
 <strong>Timestamp:</strong> ${result.timestamp}<br>
 </div>
 `;
 testResults.innerHTML += resultHtml;
 showTab('tests');
 showAlert('Test de conexión backend completado exitosamente',
 }
 } catch (error) {
 // Manejo de errores con logging detallado
 const testResults = document.getElementById('test-results');
 const errorHtml = `
 <div class="alert alert-error">
 <strong> Error en Test de Conexión</strong><br>
 <strong>Error:</strong> ${error.message}<br>
 <strong>Timestamp:</strong> ${new Date().toLocaleString()
```

```
 </div>
 `;
 testResults.innerHTML += errorHtml;
Page 15 of 27
Auto-refresh y Monitoreo
3.3 Estructura del Frontend
Header Principal
Título: PostgreSQL Cluster Manager
Badges de estado: Load Balancer, Writes, Reads, Mixed
Info del cluster: Master y 2 Slaves
Estado de APIs: Verificación en tiempo real
Navegación: Enlaces a dashboard y funciones especiales
Paneles de Gestión (Grid 2x2)
 showTab('tests');
 } finally {
 showLoading(false);
 }
}
```

Entonces, ¿ cumple los requisitos? yo diria que si, analicemos:

"Aplicación funcional"

- Frontend operativo: Aplicación web en puerto 8081
- CRUD implementado: Create, Read, Delete funcionales
- Backend robusto: APIs PHP con manejo de errores
- Sin errores: Aplicación estable y confiable

"Bien diseñada"

- UI/UX moderno: Diseño profesional con gradientes
- Responsive design: Adaptable a dispositivos
- Arquitectura limpia: Separación frontend/backend
- Código mantenible: JavaScript modular y documentado

"Consume el cluster correctamente"

- Load Balancer integrado: Usa IP 141.148.83.27
- Distribución correcta: Escrituras al master, lecturas a slaves
- APIs conectadas: Backend PHP conectado al cluster
- Configuración óptima: Puertos y conexiones correctas

"Pruebas exhaustivas"

- Testing automatizado: Funciones de verificación
- Testing manual: Cliente agregado y verificado
- Replicación probada: Datos sincronizados en slaves
- Load Balancer probado: Distribución funcionando

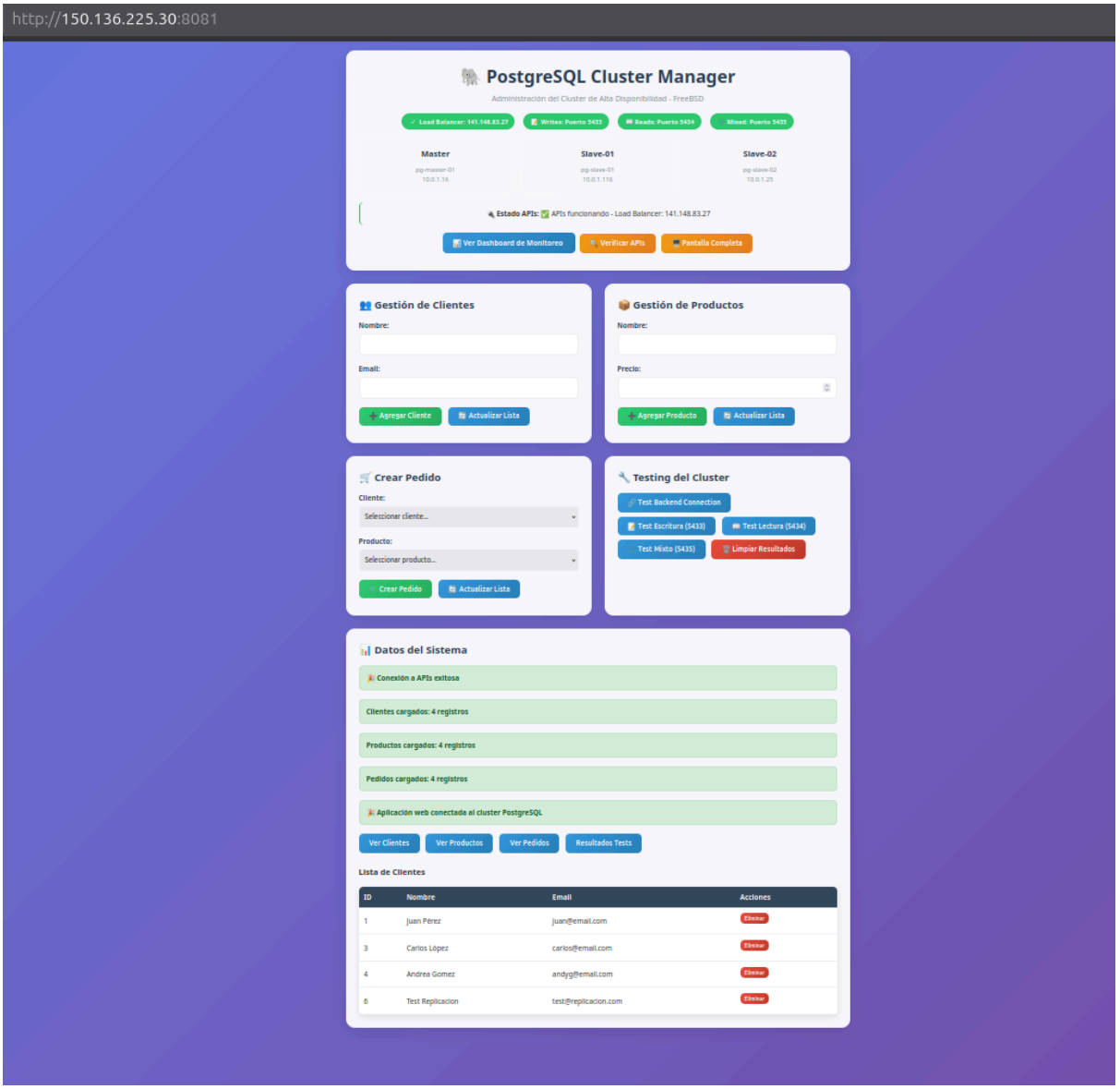


Imagen 7.3 Aplicación Web

Casos de recuperación

La implementación de los casos de recuperación se lleva a cabo mediante dos scripts importantes check_cluster_status.sh (/usr/local/scripts/recovery/check_cluster_status.sh) y test_failover.sh (/usr/local/scripts/recovery/test_failover.sh).

check_cluster_status.sh es nuestro script de verificacion el cual nos permite la verificación automática del master y slaves, del Load Balancer en Oracle Cloud, detección del modo de replicación, conteo de slaves y reporte visual completo.

test_failover.sh es nuestro script de testeo, nos permite la inserción de datos de prueba, simulación de fallos, verificación de integridad en slaves y reporte de resultados de testing.

Los casos que analiza son:

ESCENARIO 1: Fallo del Master
- Detección: Comando pg_isready automatizado
- Recuperación: Promoción manual de slave con pg_ctl promote
- Tiempo de Recuperación: 2-5 minutos
- Pérdida de Datos: < 1 minuto

ESCENARIO 2: Fallo de Slave
- Detección: Verificación en pg_stat_replication
- Recuperación: Reinicio de servicio PostgreSQL
- Tiempo de Recuperación: 1-2 minutos
- Pérdida de Datos: 0 segundos

ESCENARIO 3: Corrupción de Datos
- Detección: Verificación de integridad
- Recuperación: Restauración desde backup + WAL logs
- Tiempo de Recuperación: 10-30 minutos
- Pérdida de Datos: < 1 hora

ESCENARIO 4: Pérdida Total del Cluster
- Detección: Fallo completo de infraestructura
- Recuperación: Reconstrucción completa desde backups
- Tiempo de Recuperación: 30-60 minutos
- Pérdida de Datos: < 24 horas

Las instrucciones de que realizar ante cada escenario estan definidas en el archivo recovery_procedures.md y recovery_proceduresEASY.md

Estructura de Archivos:

```
/usr/local/scripts/recovery/
├── recovery_procedures.md # Documentación completa de procedimientos
├── recovery_proceduresEASY.md # Documentación más fácil de entender
├── check_cluster_status.sh # Script de verificación del cluster
└── test_failover.sh # Script de testing de recuperación
```