

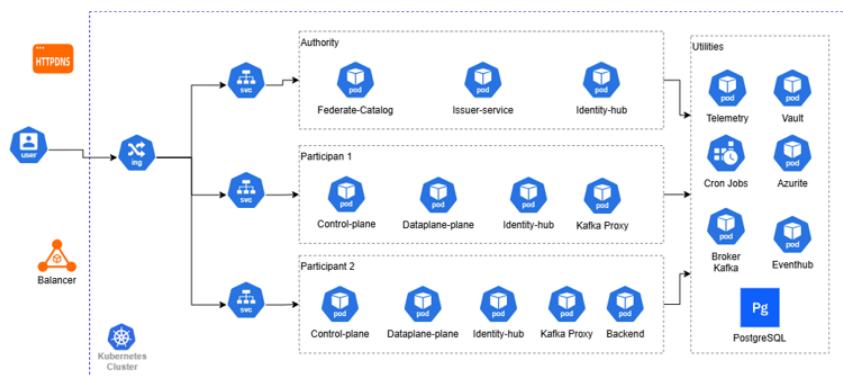
MVD - EONAX

Introducción:

Este documento detalla el proceso completo de despliegue y ejecución de los tests end-to-end del ****Dataspace Ecosystem****.

Contexto del Proyecto

El objetivo es validar el funcionamiento de un dataspace mínimo viable (MVD - Minimum Viable Dataspace) que simula el intercambio seguro de datos entre participantes en el ecosistema.



Componentes Principales

- **Authority:** Autoridad de confianza que emite credenciales
- **Provider:** Organización que ofrece datos/APIs
- **Consumer:** Organización que consume datos

Instalación de MVD

Los pasos basicos para instalación de este MVD se encuentran en el siguiente readme

[dataspace-ecosystem/system-tests/readme.md at main · AmadeusITGroup/dataspace-ecosystem](https://github.com/AmadeusITGroup/dataspace-ecosystem/blob/main/system-tests/readme.md)

Problemas:

Al ejecutar el comando

```
1 terraform -chdir=system-tests apply -auto-approve -var="environment=local"
```

el resultado era fallido, dado que los scripts de terraform buscan imágenes de docker que inician con localhost

por ejemplo `localhost/kafka-proxy-k8s-manager:latest`

Para solventar dicho problema es necesario re-taggear las imágenes construidas:

```
1 docker tag control-plane-postgresql-hashicorpvaul:latest localhost/co
2 docker tag data-plane-postgresql-hashicorpvaul:latest localhost/data-
3 docker tag federated-catalog-postgresql-hashicorpvaul:latest localhost/
4 docker tag identity-hub-postgresql-hashicorpvaul:latest localhost/ide
5 docker tag issuer-service-postgresql-hashicorpvaul:latest localhost/i
6 docker tag telemetry-service-postgresql-hashicorpvaul:latest localhost/
7 docker tag telemetry-agent-postgresql-hashicorpvaul:latest localhost/
8 docker tag backend-service-provider:latest localhost/backend-service-p
9 docker tag telemetry-storage-postgresql-hashicorpvaul:latest localhost
10 docker tag telemetry-csv-manager-postgresql-hashicorpvaul:latest loca
```

y luego es necesario cargar esas imágenes al cluster con kind, de la siguiente manera

```
1 Kind load docker-image \
2   control-plane-postgresql-hashicorpvaul:latest \
3   localhost/control-plane-postgresql-hashicorpvaul:latest \
4   data-plane-postgresql-hashicorpvaul:latest \
5   localhost/data-plane-postgresql-hashicorpvaul:latest \
6   federated-catalog-postgresql-hashicorpvaul:latest \
7   localhost/federated-catalog-postgresql-hashicorpvaul:latest \
8   identity-hub-postgresql-hashicorpvaul:latest \
9   localhost/identity-hub-postgresql-hashicorpvaul:latest \
10  issuer-service-postgresql-hashicorpvaul:latest \
11  localhost/issuer-service-postgresql-hashicorpvaul:latest \
12  telemetry-service-postgresql-hashicorpvaul:latest \
13  localhost/telemetry-service-postgresql-hashicorpvaul:latest \
14  telemetry-agent-postgresql-hashicorpvaul:latest \
15  localhost/telemetry-agent-postgresql-hashicorpvaul:latest \
16  backend-service-provider:latest \
17  localhost/backend-service-provider:latest \
18  telemetry-storage-postgresql-hashicorpvaul:latest \
19  localhost/telemetry-storage-postgresql-hashicorpvaul:latest \
20  telemetry-csv-manager-postgresql-hashicorpvaul:latest \
21  localhost/telemetry-csv-manager-postgresql-hashicorpvaul:latest \
22  federated-catalog-filter-postgresql-hashicorpvaul:latest \
23  localhost/federated-catalog-filter-postgresql-hashicorpvaul:latest \
24  kafka-proxy-entra-auth:latest \
25  localhost/kafka-proxy-entra-auth:latest \
26  kafka-proxy-k8s-manager:latest \
27  localhost/kafka-proxy-k8s-manager:latest \
28  -n dse-cluster
```

¿Por qué cargar las imágenes?

Kind es un cluster aislado. Necesita tener las imágenes dentro del cluster para poder desplegarlas.

Luego de esto, es necesario volver a realizar un apply de terraform

```
terraform -chdir=system-tests apply -auto-approve -
var="environment=local"
```

El resultado de esto es ver en estado Running o completed, los pods que estan dentro del cluster

```

dataspaces@ecosystem git:(main) kubectl get pods
NAME                                         READY   STATUS    RESTARTS   AGE
authority-db-init-gfq76                      0/1     Completed  0          31h
authority-federatedcatalog-76499894f-qs5fx   1/1     Running   0          31h
authority-federatedcatalogfilter-5b998cbc5f-nmkh8 1/1     Running   0          31h
authority-identityhub-fb7b4bc9-ht4n4        1/1     Running   1 (31h ago) 31h
authority-issuerservice-6d8c5bbcc4-w5hj5      1/1     Running   0          31h
authority-telemetrycsvmanager-55bc77db9f-g48zw 1/1     Running   0          31h
authority-telemetryservice-c6b7898b-wcdtg     1/1     Running   0          31h
authority-telemetrystorage-6484bc8564-58r6l   1/1     Running   0          31h
authority-vault-0                            1/1     Running   0          31h
azurite-0                                    1/1     Running   0          31h
azurite-report-storage-0                     1/1     Running   0          31h
billing-db-insert-hzm68                      0/1     Completed  0          31h
billinguser-db-init-kj8h7                     0/1     Completed  0          31h
broker-5f6cf9c487-prk64                      1/1     Running   0          31h
consumer-controlplane-6785b7547-tzqvc       1/1     Running   0          31h
consumer-dataplane-7fdbb7f7bc-5rkkm          1/1     Running   0          31h
consumer-db-init-4ps2d                       0/1     Completed  0          31h
consumer-identityhub-7679dd7d7c-vlvfx       1/1     Running   0          31h
consumer-kafka-proxy-k8s-manager-5bccbf6cf9-bxckn 1/1     Running   0          31h
consumer-telemetryagent-54ffcff99-2lqrt     1/1     Running   0          31h
consumer-vault-0                            1/1     Running   0          31h
create-blob-container-qn667                   0/1     Completed  0          31h
eventhubs-0                                   1/1     Running   0          31h
kafka-proxy-provider-57f9979556-wlc9b       1/1     Running   0          31h
kafka-proxy-provider-oauth2-5ffd8c568-h5j49  1/1     Running   0          31h
kafkacat-57784864bf-k7zmv                  1/1     Running   0          31h
postgresql-0                                1/1     Running   0          31h
provider-backend-5f59cd5477-qv4hr           1/1     Running   0          31h
provider-controlplane-85db574c6c-shjdd       1/1     Running   0          31h
provider-dataplane-64d7c95bc-vbbkc          1/1     Running   0          31h
provider-db-init-9scbw                      0/1     Completed  0          31h
provider-identityhub-6566d48bfc-v5lxm       1/1     Running   0          31h
provider-kafka-proxy-k8s-manager-7495cd5d8-s9lc2 1/1     Running   0          31h
provider-telemetryagent-65949f7cdf-6qlc8     1/1     Running   0          31h
provider-vault-0                            1/1     Running   0          31h
→ dataspaces@ecosystem git:(main) kubectl port-forward eventhubs-0 52717:5672
kubectl port-forward postgresql-0 57521:5432 &
Forwarding from 127.0.0.1:52717 -> 5672
Forwarding from [::1]:52717 -> 5672

```

Que despliega **terraform**:

- **Infraestructura Base**

- PostgreSQL (base de datos compartida)
- Azure Event Hub (emulador local)
- HashiCorp Vault (gestión de secretos)
- Azurite (emulador de Azure Storage)
- Kafka Broker (para streaming)

- **Por cada participante (Authority, Provider, Consumer)**

- Control Plane (gestión de contratos y catálogo)
- Data Plane (transferencia de datos)
- Identity Hub (gestión de identidades)
- Vault (instancia propia de secretos)
- Init jobs (inicialización de base de datos)

- **Componentes específicos de Authority**

- Issuer Service (emisión de credenciales)
- Federated Catalog (catálogo centralizado)
- Telemetry Service (recolección de eventos)
- Telemetry Storage (almacenamiento de telemetría)
- Telemetry CSV Manager (generación de reportes)

• Componentes específicos de Provider

- Backend Service (APIs de datos reales)
- Kafka Proxy Manager (gestión de proxies Kafka)
- Telemetry Agent (envío de telemetría)

• Componentes específicos de Consumer

- Kafka Proxy Manager (gestión de proxies Kafka)
- Telemetry Agent (envío de telemetría)

EndToEnd Test.

Los tests necesitan acceso directo a Event Hub y PostgreSQL que tenemos levantado en nuestro cluster con kind, para lo cual es necesario ejecutar los siguientes comandos:

```
1 kubectl port-forward eventhubs-0 52717:5672
2 kubectl port-forward postgresql-0 57521:5432 &
```

Posterior a aquello, es posible ejecutar test EndToEnd, con los test que estan realizados dentro del proyecto.

```
./gradlew :system-tests:runner:test -DincludeTags="EndToEndTest"
```

Esto ejecutara los test que tengan el tag `EndToEndTest` en el submodulo runner.

Esté es el resultado de una ejecución de test satisfactorios.

```
+ dataspace-ecosystem git:(main) ./gradlew :system-tests:runner:test -DincludeTags="EndToEndTest"
> Task :system-tests:runner:compileTestFixturesJava
Note: /home/itandazo/Projects/Dataspace/examples/dataspace-ecosystem/system-tests/runner/src/test/fixtures/java/org/eclipse/eeo/test/system/AbstractParticipant.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

> Task :system-tests:runner:compileTestJava
Note: /home/itandazo/Projects/Dataspace/examples/dataspace-ecosystem/system-tests/runner/src/test/java/org/eclipse/eeo/test/system/LocalEndToEndTests.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: /home/itandazo/Projects/Dataspace/examples/dataspace-ecosystem/system-tests/runner/src/test/java/org/eclipse/eeo/test/system/LocalEndToEndTests.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

> Task :system-tests:runner:test
[...]
| Results: SUCCESS (19 tests, 19 passed, 0 failed, 0 skipped)

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.
For more on this, please refer to https://docs.gradle.org/8.4/userguide/command\_line\_interface.html#sec:command\_line\_warnings in the Gradle documentation.

BUILD SUCCESSFUL in 3s
24 actionable tasks: 5 executed, 19 up-to-date
+ dataspace-ecosystem git:(main) ./gradlew :system-tests:runner:test -DincludeTags="EndToEndTest"
```

Si se desea ver logs de la ejecucion de los test, es posible hacerlo asi

```
# Logs del consumer (negociación de contratos)
```

```
kubectl logs -f deployment/consumer-controlplane
```

```
# Logs del provider (transferencia de datos)
```

```
kubectl logs -f deployment/provider-dataplane
```

```
# Logs de telemetría
```

```
kubectl logs -f deployment/authority-telemetryservice
```

Los tests simulan un **flujo completo de intercambio de datos** entre tres participantes de un dataspace, validando desde el descubrimiento de datos hasta la generación de reportes de telemetría.

FASE 1: Setup Inicial (@BeforeAll)

¿Qué hace?

Prepara todo el ecosistema antes de ejecutar cualquier test.

Pasos principales:

1. Registrar participantes en la Authority

- Crea 3 participantes: Authority, Provider, Consumer
- Cada uno con su identidad descentralizada (DID)

2. Emitir credenciales verificables

- MembershipCredential: Prueba que perteneces al dataspace
- DomainCredential: Prueba que perteneces a un dominio específico (route/travel)
- Cada participante recibe 2 credenciales

3. Publicar 11 assets en el Provider

- APIs REST (básicas, con OAuth2, con restricciones)
- Streams de Kafka
- Cada asset tiene:
 - Metadata (nombre, descripción)
 - DataAddress (dónde están los datos reales)
 - Policy (quién puede acceder)

Resultado: El ecosistema listo con participantes autenticados y datasets disponibles.

FASE 2: Tests de Catálogo (3 tests)

¿Qué validan?

Que los participantes pueden **descubrir** qué datos están disponibles en el dataspace.

Tests:

1. catalog_provider()

- 1 ✓ La Authority consulta el catálogo del Provider
- 2 ✓ Valida que los 11 assets están disponibles
- 3 ✓ Verifica que cada asset tiene timestamp de creación

2. catalog_consumer()

- 1 ✓ El Consumer consulta su propio catálogo
- 2 ✓ Valida que está vacío (el consumer no publica datos)

3. catalog_consumer_restricted()

- 1 ✓ El Consumer consulta el catálogo del Provider
- 2 ✓ Solo ve assets del dominio "route" (porque tiene DomainCredential rout

```
3 ✘ NO ve assets del dominio "travel" (no tiene esa credencial)
4 ✓ Valida que el descubrimiento está restringido por credenciales
```

Resultado: El descubrimiento de catálogo funciona y respeta las políticas de visibilidad.

FASE 3: Tests de Transferencia (11 tests)

¿Qué validan?

Que los participantes pueden **negociar contratos** y **transferir datos** de forma segura.

Flujo general de una transferencia:

- 1 1. Descubrir asset en catálogo
- 2 2. Negociar contrato (validar políticas y credenciales)
- 3 3. Iniciar transferencia (obtener token de acceso)
- 4 4. Consumir datos (usando el token)
- 5 5. Enviar telemetría

Tests principales:

1. transfer_success() - 4 variaciones

- 1 ✓ Transfiere datos de API REST básica
- 2 ✓ Transfiere datos de API con restricción de dominio
- 3 ✓ Transfiere datos de API con OAuth2
- 4 ✓ Transfiere datos de API con parámetros embebidos
- 5 ✓ Valida que los datos recibidos son correctos

2. transfer_kafka_stream_oauth()

- 1 ✓ Negocia contrato para stream Kafka con OAuth2
- 2 ✓ Despliega proxy Kafka dinámicamente
- 3 ✓ Finaliza la transferencia correctamente

3. transfer_kafka_stream()

- 1 ✓ Negocia contrato para stream Kafka
- 2 ✓ Despliega proxy Kafka automáticamente en Kubernetes
- 3 ✓ Provider publica mensaje: "Hello from provider!"
- 4 ✓ Consumer lee el mensaje a través del proxy
- 5 ✓ Valida que el mensaje llegó correctamente
- 6 ✓ Limpia el proxy después de la transferencia

4. transfer_failure()

- 1 ✓ Intenta transferir datos de una API que falla (500 error)
- 2 ✓ Valida que el error se maneja correctamente
- 3 ✓ Verifica mensaje de error apropiado

5. transfer_whenContractExpiration_shouldTerminateTransferProcessAtExpiration()

- 1 ✓ Transfiere datos de una API con contrato que expira en 20 segundos
- 2 ✓ Valida que los datos son accesibles inicialmente
- 3 ⏱ Espera 25 segundos
- 4 ✓ Valida que el acceso se bloquea automáticamente
- 5 ✓ Verifica que el proceso de transferencia se termina (TERMINATED)

6. transfer_retrieveAgreement_shouldBlockFurtherAccess()

- 1 ✓ Transfiere datos exitosamente
- 2 ✓ Provider retira el contrato manualmente
- 3 ✓ Valida que el acceso se bloquea inmediatamente
- 4 ✓ Provider reactiva el contrato
- 5 ✓ Valida que el acceso se restaura

- 6 ✓ Verifica el ciclo completo de retiro/reactivación
- 7. transfer_forRestrictedDiscoveryAssets()
 - 1 ✓ Transfiere datos de asset con descubrimiento restringido
 - 2 ✓ Consumer tiene la credencial correcta (domain=route)
 - 3 ✓ Valida que la transferencia es exitosa
- 8. transfer_forRestrictedDiscoveryAssets_NotAvailable()
 - 1 ✓ Intenta transferir asset con descubrimiento restringido
 - 2 ✗ Consumer NO tiene la credencial correcta (necesita domain=travel)
 - 3 ✓ Valida que la negociación se termina (TERMINATED)
- 9. transfer_whenPolicyNotMatched_shouldTerminate()
 - 1 ✓ Intenta negociar contrato con policy imposible de satisfacer
 - 2 ✓ Valida que la negociación se rechaza automáticamente
 - 3 ✓ Verifica estado TERMINATED

Resultado: El sistema negocia contratos, valida políticas, transfiere datos HTTP y Kafka, y gestiona errores correctamente.

FASE 4: Tests de Reportes (4 tests)

¿Qué validan?

Que el sistema recolecta **telemetría** de las transferencias y genera **reportes CSV** mensuales.

Tests:

- 1. testReportGenerationSucceeds()
 - 1 ✓ Crea eventos de telemetría simulados (consumer y provider)
 - 2 ✓ Sigue generación de reporte CSV para septiembre 2025
 - 3 ✓ Recupera el reporte usando autenticación JWT
 - 4 ✓ Valida formato del CSV:
 - contract_id
 - data_transfer_response_status (200)
 - total_transfer_size_in_kB (0.02)
 - total_number_of_events (1)
 - 5 ✓ Valida que funciona con JWTs con múltiples roles
- 2. testReportGenerationWithOnlyOnePartySucceeds()
 - 1 ✓ Crea evento de telemetría solo del consumer (sin provider)
 - 2 ✓ Genera reporte exitosamente con datos parciales
 - 3 ✓ Valida que el sistema es robusto ante datos incompletos
- 3. testRetrieveReportWithNonExistentParticipantFails()
 - 1 ✓ Intenta obtener reporte con JWT de participante inexistente
 - 2 ✓ Valida que retorna 403 Forbidden
 - 3 ✓ Valida seguridad del sistema
- 4. testRetrieveNonExistentReportFromExistentParticipantFails()
 - 1 ✓ Intenta obtener reporte que no existe
 - 2 ✓ Valida que retorna 404 Not Found
 - 3 ✓ Valida manejo correcto de errores

Resultado: El sistema recolecta telemetría, genera reportes mensuales y protege el acceso con autenticación.

FASE 5: Cleanup (@AfterAll)

¿Qué hace?

Procesa eventos de telemetría pendientes y valida que todo se guardó correctamente.

Pasos:

- 1 1. Conecta a Event Hub
- 2 2. Consume todos los eventos de telemetría generados durante los tests
- 3 3. Envía cada evento a la Authority Telemetry Service
- 4 4. Valida que los datos se guardaron en PostgreSQL
- 5 5. Verifica que todos los contratos fueron procesados (timeout: 1 minuto)

Resultado: Todos los eventos de telemetría están almacenados y listos para generar reportes.

Componentes Involucrados

Componente	Rol en los Tests
authority-identityhub	Almacena identidades de participantes
authority-issuerservice	Emite credenciales verificables
authority-federatedcatalog	Recolecta catálogos de todos los participantes
authority-telemetryservice	Recibe eventos de telemetría
authority-telemetrycsvmanager	Genera reportes CSV mensuales
provider-controlplane	Publica assets y negocia contratos
provider-dataplane	Transfiere datos reales
provider-backend	APIs y servicios de datos reales
consumer-controlplane	Descubre datos y negocia contratos
consumer-dataplane	Consume datos transferidos
PostgreSQL	Almacena todos los datos persistentes
Event Hub	Cola de eventos de telemetría
Kafka	Streaming de datos en tiempo real
Vault	Gestión de secretos y claves