How-To Guides

Table of Contents

Find a Procedure . . . 2
Run OpenDJ in Kubernetes . . . 3
  Create the Service and StatefulSet . . . 3
  Verify the Directory and Persistence . . . 5
Set Up OpenDJ With a BCFKS Keystore . . . 6
  Create the Keystore . . . 6
  Configure and Start OpenDJ . . . 7
Use a Relational Database as an LDAP Backend . . . 8
  Start PostgreSQL . . . 8
  Create the LDAP Backend . . . 8
Run Embedded LDAP in a Spring Boot Application . . . 10
  Add the Dependency . . . 10
  Configure the Bean and Import Data . . . 10
  Verify Through LDAP . . . 11
Access Directory Entries Over REST . . . 13
  Enable the Handler . . . 13
  Read a User . . . 13
Delegate Password Reset Permissions . . . 15
  Create the Delegate and Grant Access . . . 15
  Verify the Scope . . . 15
Generate and Load Test LDIF Data . . . 17
  Generate the File . . . 17
  Add and Verify Entries . . . 17
Store Directory Data in Cassandra . . . 18
Attach a Java Debugger to a Local Instance . . . 19

***Practical recipes for running and integrating OpenDJ. These guides are** maintained in the OpenDJ repository; the former wiki articles link here. For procedures already covered by another guide, see the task directory below.*

Find a Procedure

The examples use a disposable local instance and illustrative passwords. Run server commands from the installation directory. Follow the installation prerequisites, including Java 11 or later, before starting.

Task	Procedure
Download and set up a server, with an empty suffix or sample data	Command-line installation
Stop, remove and reinstall a disposable server	Uninstalling OpenDJ; then follow command-line installation. Back up any data to keep before removal.
Check server status	Monitoring OpenDJ
Back up and restore data	Backing Up and Restoring Data
Enable and initialize replication, or remove a replica	Managing Data Replication
Inspect indexes and rebuild degraded indexes	Managing Indexes
List supported TLS protocols and cipher suites, or restrict them	TLS Protocols and Cipher Suites
Use an existing certificate or create a self-signed certificate	Preparing For Secure Communications
Keep Docker data when replacing a container, including with Compose	Running OpenDJ in Docker
Pass a PKCS#12 certificate to the Docker image	Docker image configuration (`OPENDJ_SSL_OPTIONS`)
Use OpenDJ with Jira	Atlassian's integration instructions

The following chapters cover additional tasks. Each procedure has one maintained location; use the links above for the full instructions rather than a second copy of them.

Run OpenDJ in Kubernetes

This recipe runs one directory server with persistent storage in a local Minikube cluster. It requires `minikube`, `kubectl`, and a default StorageClass that supports dynamic volume provisioning. It does not configure replication; increasing the replica count alone does not create a replicated directory.

```
minikube start --cpus=4 --memory=8g
kubectl create namespace opendj-demo
kubectl -n opendj-demo create secret generic opendj-root \
  --from-literal=password='DemoPassword123!'
```

Create the Service and StatefulSet

Save the following as `opendj.yaml`. The root password comes from the Secret. A PersistentVolumeClaim keeps the instance under `/opt/opendj/data`, the same location described in Running OpenDJ in Docker. `fsGroup` lets the non-root server write to the mounted volume on storage that supports Kubernetes volume ownership management.

```
apiVersion: v1
kind: Service
metadata:
  name: opendj
spec:
  clusterIP: None
  selector:
    app: opendj
  ports:
    - name: ldap
      port: 1389
    - name: ldaps
      port: 1636
    - name: admin
      port: 4444
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: opendj
spec:
  serviceName: opendj
  replicas: 1
  selector:
    matchLabels:
      app: opendj
  template:
    metadata:
      labels:
```

```
        app: opendj
    spec:
      terminationGracePeriodSeconds: 60
      securityContext:
        fsGroup: 1001
      containers:
        - name: opendj
          image: openidentityplatform/opendj:5.2.0
          env:
            - name: ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: opendj-root
                  key: password
            - name: ADD_BASE_ENTRY
              value: --addBaseEntry
          ports:
            - containerPort: 1389
            - containerPort: 1636
            - containerPort: 4444
          readinessProbe:
            exec:
              command: [/opt/opendj/healthcheck.sh]
            periodSeconds: 10
            timeoutSeconds: 30
          volumeMounts:
            - name: data
              mountPath: /opt/opendj/data
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: [ReadWriteOnce]
        resources:
          requests:
            storage: 1Gi
```

```
kubectl -n opendj-demo apply -f opendj.yaml
kubectl -n opendj-demo rollout status statefulset/opendj --timeout=600s
kubectl -n opendj-demo get pods,pvc
```

Wait for the pod to be Ready. The readiness probe checks completed bootstrap and LDAP availability; a listening TCP port alone is insufficient during first startup. If the claim stays Pending, check the StorageClass. If the pod does not become Ready, inspect `kubectl -n opendj-demo logs opendj-0` and `kubectl -n opendj-demo describe pod opendj-0`.

Verify the Directory and Persistence

```
kubectl -n opendj-demo exec opendj-0 -- /opt/opendj/bin/ldapsearch \
  --hostname localhost --port 1389 \
  --bindDN 'cn=Directory Manager' --bindPassword 'DemoPassword123!' \
  --baseDN dc=example,dc=com --searchScope base '(objectClass=*)' dn
```

Expect `dn: dc=example,dc=com`. Entries added to this instance remain on the claim when the pod is replaced. To verify, add a test entry, delete only the pod with `kubectl -n opendj-demo delete pod opendj-0`, wait for the replacement to become Ready, and search for the entry again. Do not delete the claim during this check.

Deleting the StatefulSet normally retains its claim. Deleting the demonstration namespace with `kubectl delete namespace opendj-demo` also deletes its claims and can delete the backing storage, depending on its reclaim policy.

Set Up OpenDJ With a BCFKS Keystore

This recipe creates a BCFKS keystore and uses it for a fresh OpenDJ server. It demonstrates Bouncy Castle FIPS provider registration, not certification of a deployment or configuration of an operating system's FIPS mode. Use the provider shipped with your OpenDJ distribution, rather than downloading the old library versions from the wiki example.

Prepare a fresh unpacked installation using the installation prerequisites. Run the following commands from its root directory with the same Java installation that will run OpenDJ. The example uses a self-signed certificate for `localhost`; see Preparing For Secure Communications for CA-signed certificates.

Create the Keystore

Store the keystore outside the installation so setup does not overwrite it. Replace the example password before using the procedure outside a disposable test.

```
umask 077
mkdir -p ../opendj-certs
CERT_DIR="$(cd ../opendj-certs && pwd)"
printf '%s\n' 'KeystorePassword123!' > "$CERT_DIR/keystore.pin"

keytool -genkeypair -alias server-cert -keyalg RSA -keysize 2048 \
  -sigalg SHA256withRSA -dname 'CN=localhost' -ext 'SAN=dns:localhost' \
  -validity 365 -storetype BCFKS -keystore "$CERT_DIR/server.bcfks" \
  -providerclass org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider \
  -providerpath lib/org.bouncycastle.bc-fips.jar \
  -storepass:file "$CERT_DIR/keystore.pin" -keypass:file "$CERT_DIR/keystore.pin"

keytool -genkeypair -alias admin-cert -keyalg RSA -keysize 2048 \
  -sigalg SHA256withRSA -dname 'CN=localhost' -ext 'SAN=dns:localhost' \
  -validity 365 -storetype BCFKS -keystore "$CERT_DIR/server.bcfks" \
  -providerclass org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider \
  -providerpath lib/org.bouncycastle.bc-fips.jar \
  -storepass:file "$CERT_DIR/keystore.pin" -keypass:file "$CERT_DIR/keystore.pin"

keytool -list -storetype BCFKS -keystore "$CERT_DIR/server.bcfks" \
  -providerclass org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider \
  -providerpath lib/org.bouncycastle.bc-fips.jar \
  -storepass:file "$CERT_DIR/keystore.pin"
```

The listing should report `BCFKS`, provider `BCFIPS`, and two private-key entries: `server-cert` for LDAP TLS and `admin-cert` for the administration connector. Pass both aliases to setup so its truststores include both certificates and it can verify startup through the administration connector. The JAR has a stable distribution filename even when its bundled version changes.

Configure and Start OpenDJ

Register the provider in each environment used to start the server or its tools:

```
export OPENDJ_JAVA_ARGS="${OPENDJ_JAVA_ARGS:-}
-Dorg.openidentityplatform.opendj.fips.register=true"

./setup --cli --no-prompt --acceptLicense \
  --hostname localhost --ldapPort 1389 --ldapsPort 1636 --adminConnectorPort 4444 \
  --enableStartTLS --useBcfksKeystore "$CERT_DIR/server.bcfks" \
  --keyStorePasswordFile "$CERT_DIR/keystore.pin" \
  --certNickname server-cert --certNickname admin-cert \
  --rootUserDN 'cn=Directory Manager' --rootUserPassword password \
  --baseDN dc=example,dc=com --addBaseEntry

bin/ldapsearch --hostname localhost --port 1636 --useSSL --trustAll \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --baseDN dc=example,dc=com --searchScope base '(objectClass=*)' dn
```

The search should return the base entry over LDAPS. Also verify the administration connector:

```
bin/dsconfig list-backends --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password --trustAll --no-prompt
```

Expect the enabled `userRoot` backend. `--trustAll` is only used to check this local self-signed installation. Configure certificate trust for normal client use; do not replace a system truststore merely to make this example work. Preserve the provider setting in the service environment for subsequent starts.

Use a Relational Database as an LDAP Backend

This example stores LDAP entries in PostgreSQL while clients continue to use LDAP. It requires Docker and a fresh OpenDJ installation from the full server ZIP, prepared according to the installation prerequisites. Use Java 11 or later.

Start PostgreSQL

```
docker run -d --name opendj-postgres \
  -p 127.0.0.1:5432:5432 \
  -e POSTGRES_DB=opendj -e POSTGRES_PASSWORD=password \
  postgres:17

docker exec opendj-postgres pg_isready -U postgres -d opendj
```

Wait until `pg_isready` reports that the database accepts connections. This container is a disposable demonstration; configure database persistence and credentials for a lasting deployment.

Create the LDAP Backend

Run setup from the unpacked OpenDJ directory without a base DN, so it does not create `userRoot`:

```
./setup --cli --no-prompt --acceptLicense \
  --hostname localhost --ldapPort 1389 --adminConnectorPort 4444 \
  --enableStartTLS --generateSelfSignedCertificate \
  --rootUserDN 'cn=Directory Manager' --rootUserPassword password

bin/dsconfig create-backend --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --backend-name userRoot --type jdbc \
  --set base-dn:dc=example,dc=com \
  --set 'db-
directory:jdbc:postgresql://localhost:5432/opendj?user=postgres&password=password' \
  --set enabled:true --trustAll --no-prompt
```

Quote the JDBC URL so the shell does not interpret `&`. The example's database credentials are stored in the backend configuration; restrict access to the instance files. The PostgreSQL driver is included in the full server ZIP; `opendj-<version>-slim.zip` leaves out the Cassandra, Oracle, MySQL, SQL Server and PostgreSQL drivers. For another supported database, check the backend's SQL dialect support as well as the JDBC driver; a JDBC driver alone does not make every database compatible.

The backend is initially empty. Follow Generate and Load Test LDIF Data to populate it and verify LDAP searches. See dsconfig create-backend for configuration properties.

When finished, stop OpenDJ with `bin/stop-ds`, then remove the demonstration database with `docker rm -fv opendj-postgres`. The `-v` option also removes its anonymous data volume and discards the demonstration database.

Run Embedded LDAP in a Spring Boot Application

Use `opendj-embedded` for an LDAP server owned by your application's lifecycle, for example in integration tests. Each `EmbeddedOpenDJ` instance creates a temporary installation and deletes it on `close()`. Use a standalone server when directory data must survive application restarts. Run one embedded server at a time in a JVM, with its LDAP, administration and JMX ports available.

Add the Dependency

In an existing Spring Boot Maven application, add:

```
<dependency>
    <groupId>org.openidentityplatform.opendj</groupId>
    <artifactId>opendj-embedded</artifactId>
    <version>5.2.0</version>
</dependency>
```

Use a JDK supported by your Spring Boot version and at least Java 11 for OpenDJ.

Configure the Bean and Import Data

Put this configuration in your application's component-scan package. It imports `src/main/resources/test-directory.ldif` after startup and calls the server's `close()` method when the Spring context closes or initialization fails:

```
import java.io.IOException;
import java.io.InputStream;
import org.openidentityplatform.opendj.embedded.Config;
import org.openidentityplatform.opendj.embedded.EmbeddedOpenDJ;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class LdapConfiguration {
    @Bean(destroyMethod = "close")
    public EmbeddedOpenDJ embeddedOpenDJ(
            @Value("${opendj.password:TestPassword123!}") String password) throws
Exception {
        Config config = new Config();
        config.setBaseDN("dc=example,dc=com");
        config.setAdminPassword(password);
        config.setPort(1389);
        config.setAdminPort(4444);
        config.setJmxPort(1689);
```

```
        EmbeddedOpenDJ server = new EmbeddedOpenDJ(config);
        try {
            server.run();
            try (InputStream data = getClass().getResourceAsStream("/test-
directory.ldif")) {
                if (data == null) {
                    throw new IOException("Missing test-directory.ldif");
                }
                server.importData(data);
            }
            return server;
        } catch (Exception e) {
            server.close();
            throw e;
        }
    }
}
```

Use this LDIF file. `importData` adds entries; it does not replace existing entries. Check the logs for rejected additions and verify the imported entry:

```
dn: dc=example,dc=com
objectClass: domain
dc: example

dn: ou=people,dc=example,dc=com
objectClass: organizationalUnit
ou: people

dn: uid=jdoe,ou=people,dc=example,dc=com
objectClass: inetOrgPerson
cn: John Doe
sn: Doe
uid: jdoe
```

Verify Through LDAP

With the Spring application running, use `ldapsearch` from an OpenDJ distribution:

```
bin/ldapsearch --hostname localhost --port 1389 \
  --bindDN 'cn=Directory Manager' --bindPassword 'TestPassword123!' \
  --baseDN 'uid=jdoe,ou=people,dc=example,dc=com' --searchScope base \
  '(objectClass=*)' uid cn
```

Expect `uid: jdoe` and `cn: John Doe`. Use the same assertion in your application's integration tests, for example with Spring LDAP's `LdapTemplate`. Closing the context calls `close()` to stop the server and clean up its temporary files. If the server cannot be stopped, the instance stays open and

retains its directory and shutdown hook so that a later `close()` call or JVM exit can retry the stop.

Access Directory Entries Over REST

This recipe enables the built-in HTTP handler and reads a user through the default `/api` mapping. Start with a fresh server using command-line installation. For this recipe, pass `--hostname localhost`, `--sampleData 1000`, `--baseDN dc=example,dc=com`, `--enableStartTLS` and `--generateSelfSignedCertificate` to setup. Use administration port 4444 and LDAP port 1389, with Directory Manager password `password`. The generated certificate and its `localhost` identity are used by the HTTPS and StartTLS checks below.

Enable the Handler

Bind HTTPS to loopback and reuse the JKS certificate generated during setup. Password-reset actions require a secure connection, even on loopback. See Setting Up an HTTP Connection Handler for other certificate configurations. `--trustAll` below is for the local server's self-signed administration certificate.

```
bin/dsconfig set-connection-handler-prop \
  --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --handler-name 'HTTP Connection Handler' \
  --set listen-address:127.0.0.1 --set listen-port:8443 \
  --set use-ssl:true --set key-manager-provider:JKS --set enabled:true \
  --trustAll --no-prompt

bin/stop-ds --restart

bin/dsconfig list-http-authorization-mechanisms \
  --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password --trustAll --no-prompt

bin/dsconfig list-http-endpoints \
  --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password --trustAll --no-prompt
```

Restarting applies the listener and TLS settings. The default `/api` endpoint is enabled and uses `HTTP Basic`. The exact-match identity mapper resolves a username to its `uid` entry. The sample users have password `password`.

Read a User

Export the locally generated certificate so `curl` can verify it:

```
keytool -exportcert -rfc -alias server-cert -keystore config/keystore \
  -storepass:file config/keystore.pin -file server-cert.pem
```

```
curl --fail-with-body --cacert server-cert.pem --user 'user.0:password' \
  'https://localhost:8443/api/users/user.0?_prettyPrint=true'
```

The JSON response identifies `user.0` and includes its mapped name and contact information. The mapping is in `config/rest2ldap/endpoints/api/example-v1.json`; its default user base is `ou=people,dc=example,dc=com`. Adapt that mapping if you use a different suffix.

To enable HTTP access logging:

```
bin/dsconfig set-log-publisher-prop \
  --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --publisher-name 'File-Based HTTP Access Logger' --set enabled:true \
  --trustAll --no-prompt
```

See Performing RESTful Operations for creating, updating and deleting resources, API versions and password actions. HTTP authentication does not grant additional LDAP privileges or access rights. For read/write delegation, use Configuring Privileges and Access Control rather than relying on the name or location of a user entry. To delegate password resets, follow Delegate Password Reset Permissions.

Delegate Password Reset Permissions

Resetting another user's password requires both the `password-reset` privilege and permission to write that entry's `userPassword` attribute. This recipe grants only that write permission below `ou=People,dc=example,dc=com`.

Use a disposable server containing the sample users from Access Directory Entries Over REST. Run these commands before changing the sample password of `user.0`. `--trustAll` accepts the demonstration server's self-signed certificate.

Create the Delegate and Grant Access

Create a delegate outside the target People subtree, so its own entry is not in the delegated reset scope:

```
bin/ldapmodify --hostname localhost --port 1389 --useStartTLS --trustAll \
  --bindDN 'cn=Directory Manager' --bindPassword password <<'LDIF'
dn: ou=Administrators,dc=example,dc=com
changetype: add
objectClass: organizationalUnit
ou: Administrators

dn: uid=password-admin,ou=Administrators,dc=example,dc=com
changetype: add
objectClass: inetOrgPerson
cn: Password Administrator
sn: Administrator
uid: password-admin
userPassword: AdminPassword123!
ds-privilege-name: password-reset

dn: ou=People,dc=example,dc=com
changetype: modify
add: aci
aci: (targetattr="userPassword")(version 3.0; acl "Delegate password resets in
People"; allow (write) userdn="ldap:///uid=password-
admin,ou=Administrators,dc=example,dc=com";)
LDIF
```

The ACI is inherited by entries below People. It does not grant writes to other attributes, permission to add or delete entries, or password resets outside that subtree. Existing ACIs still apply; this example assumes the default access-control configuration.

Verify the Scope

Reset a sample user's password:

```
bin/ldappasswordmodify --hostname localhost --port 1389 --useStartTLS --trustAll \
  --bindDN 'uid=password-admin,ou=Administrators,dc=example,dc=com' \
  --bindPassword 'AdminPassword123!' \
  --authzID 'dn:uid=user.0,ou=People,dc=example,dc=com' \
  --newPassword 'NewPassword123!'
```

The password-modify operation should succeed. Verify the new password by binding as the user:

```
bin/ldapsearch --hostname localhost --port 1389 --useStartTLS --trustAll \
  --bindDN 'uid=user.0,ou=People,dc=example,dc=com' \
  --bindPassword 'NewPassword123!' \
  --baseDN 'uid=user.0,ou=People,dc=example,dc=com' --searchScope base \
  '(objectClass=*)' uid
```

Check that the delegate cannot change another attribute:

```
bin/ldapmodify --hostname localhost --port 1389 --useStartTLS --trustAll \
  --bindDN 'uid=password-admin,ou=Administrators,dc=example,dc=com' \
  --bindPassword 'AdminPassword123!' <<'LDIF'
dn: uid=user.0,ou=People,dc=example,dc=com
changetype: modify
replace: description
description: This change must be rejected
LDIF
```

Expect LDAP result 50, `Insufficient Access Rights`. A successful change indicates that another ACI grants broader access.

If the loopback HTTPS handler from the REST recipe is enabled, the same delegate can request a generated password through REST:

```
curl --fail-with-body --cacert server-cert.pem --user 'password-
admin:AdminPassword123!' \
  --request POST 'https://localhost:8443/api/users/user.0?_action=resetPassword'
```

The response contains `generatedPassword`; it replaces the password set above. See Resetting Passwords for the API details.

Generate and Load Test LDIF Data

Use `makeldif` to generate repeatable sample data, then add it through LDAP. This works with a running backend, including the JDBC backend in Use a Relational Database as an LDAP Backend. For offline bulk import and its replacement semantics, use Importing Data instead.

Generate the File

Use an empty backend with base DN `dc=example,dc=com`: do not populate it with `--addBaseEntry` or `--sampleData` during setup. The generated file includes the base entry, `ou=People`, and 100 users. See Generating Test Data to customize the template.

```
bin/makeldif --outputLDIF test.ldif --randomSeed 0 \
  --constant suffix=dc=example,dc=com --constant numusers=100 \
  config/MakeLDIF/example.template
```

Expect 102 generated entries. Change `numusers` for a larger test; the unmodified template defaults to 10,000 users. The file contains sample passwords; use it for a disposable test directory.

Add and Verify Entries

With the server running and StartTLS enabled on port 1389:

```
bin/ldapmodify --hostname localhost --port 1389 --useStartTLS --trustAll \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --defaultAdd --filename test.ldif

bin/ldapsearch --hostname localhost --port 1389 --useStartTLS --trustAll \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --baseDN 'ou=People,dc=example,dc=com' \
  '(|(uid=user.0)(uid=user.1)(uid=user.2))' uid cn
```

The search returns three entries. Unlike offline import, LDAP add does not replace the existing database: repeating the load reports `Entry Already Exists`. Use a fresh backend for another complete load. `--trustAll` is for the self-signed certificate in this local example.

Store Directory Data in Cassandra

This recipe preserves the Cassandra backend setup from the wiki's task collection. It requires a reachable Cassandra cluster and a data center name matching that cluster's configuration. Prepare a fresh OpenDJ installation from the full server ZIP (not the slim ZIP, which omits the Cassandra driver), without a base DN, using only the `./setup` command in the JDBC setup procedure. Do not run its `create-backend` command: the Cassandra backend below will use the same `userRoot` name.

Setup starts the server without `OPENDJ_JAVA_ARGS`. After setup finishes, append the driver settings to every `java-args` value in `config/java.properties` and restart the server before creating the backend. The server and offline tools such as `import-ldif --offline` and `rebuild-index` each read their own Java arguments. For a local Cassandra node on port 9042 in data center `datacenter1`:

```
unset OPENDJ_JAVA_ARGS
sed -i.bak 's/^[a-z0-9.-]*java-args=.*/& -Ddatastax-java-driver.basic.contact
-points.0=localhost:9042 -Ddatastax-java-driver.basic.load-balancing-policy.local
-datacenter=datacenter1/' config/java.properties
bin/stop-ds
bin/start-ds
```

The start scripts ignore the Java arguments in `config/java.properties` while `OPENDJ_JAVA_ARGS` is set. Leave it unset in shells and services that start the server or run offline tools. Configure authentication and TLS in the driver for a secured cluster. Create the backend with keyspace name `ldap_example_com`:

```
bin/dsconfig create-backend --hostname localhost --port 4444 \
  --bindDN 'cn=Directory Manager' --bindPassword password \
  --backend-name userRoot --type cas \
  --set db-directory:ldap_example_com --set base-dn:dc=example,dc=com \
  --set enabled:true --trustAll --no-prompt
```

Follow Generate and Load Test LDIF Data to add entries and check LDAP searches. Choose Cassandra replication and durability settings for your topology before using this backend for persistent directory data; this local recipe does not configure a production cluster.

Attach a Java Debugger to a Local Instance

Stop the disposable instance with `bin/stop-ds`, then start it with a JDWP listener on the loopback interface:

```
export OPENDJ_JAVA_ARGS="${OPENDJ_JAVA_ARGS:-}
-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=127.0.0.1:9000"
bin/start-ds --nodetach
```

Attach the IDE's remote Java debugger to `localhost:9000`. With `suspend=y`, startup waits for the debugger before the server opens its LDAP listener. Set `suspend=n` if startup should proceed without a debugger. Run clients from a second terminal without the JDWP setting, otherwise they try to bind the same debug port.

After debugging, stop the foreground server and remove the JDWP option from `OPENDJ_JAVA_ARGS` before starting it normally. Keep this unauthenticated debug listener local; it grants control of the JVM to anyone who can connect.