ORACLE REST DATA SERVICES

# ORDS in Production

Field Notes from Running Oracle REST Data Services

| POOLS | PROCESSES | DEBUG |
| --- | --- | --- |
| 2 / 0 | 300/600 | false |
| valid / invalid | | production |

| ACCESS LOG | INSTANCE API | ORA-12516 |
| --- | --- | --- |
| on | 200 | 0 |
| own folder | /status | watch |

Cinnamons Consulting Services LLC

FIRST EDITION

# ORDS in Production

Field Notes from Running Oracle REST Data Services

FIRST EDITION

Cinnamons Consulting Services LLC

# Contents


Preface 6

Introduction 7

Conventions Used in This Book 9

**Part I — Running ORDS** 10

1 One Product, Three Ways to Run It 11

2 The Configuration Folder After 22.1 15

3 Running ORDS as a Service 20

4 Upgrading ORDS Without Surprises 26

5 ORDS Behind a Reverse Proxy 30

**Part II — Configuring and Securing ORDS** 35

6 Living with the ords Command Line 36

7 Connection Pools, Sized for the Real Database 40

8 Users, Wallets and Secrets 45

**Part III — When It Breaks** 50

9 Anatomy of an Outage 51

10 Reading ORA Errors from the ORDS Side 58

11 Logs That Help Instead of Drown 64

**Part IV — Watching ORDS** 69

12 The Instance API, End to End 70

13 Signals That Predict an Outage 75

14 Minimal Monitoring Without Buying Anything 80

15 Running ORDS for Several Teams 87

A Version Field Notes, 22.1 to 26.x 92

B Checklists 94

C Application Code Against Oracle: Node.js Lessons 96

D Further Reading ..... 97
Glossary ..... 98
About Cinnamons Consulting Services LLC ..... 99

CHAPTER 9

# Anatomy of an Outage

## The Outage

The ORDS instance was down, and nobody had changed anything risky. Someone turned on a monitoring feature, ORDS needed a restart to pick it up, and the restart took the instance down.

No single setting caused it. Three ordinary settings met at the wrong moment:

1. Connection pools sized far beyond what the database could serve.
2. A restart where the old pool had not let go before the new one started.
3. Debug output left on, so the logs filled with stack traces exactly when we needed to read them.

Any one of these alone is a slow-burning risk. Together they turned a routine restart into an outage. This chapter follows the seven minutes in order, then takes each cause apart and ends with five safeguards that would have stopped it.

The setting: one standalone ORDS instance in front of Oracle Database, on a single Linux host also running other services. It is a small setup, but every number here applies to larger ones, because the limits are ratios rather than sizes.

## What happened, in order

| Stage | What we saw | What was really going on |
|---|---|---|
| Before | ORDS healthy for weeks | Pools set to `InitialLimit` 100 and `MaxLimit` 1000, but quiet traffic never pushed them |
| The change | The Instance API was turned on and a restart issued | Settings that load at startup need a restart, so this part was normal |

| Stage | What we saw | What was really going on |
|---|---|---|
| Restart | Startup looked normal at first | The old process still held its database sessions while the new one opened 100 more |
| Failure | Requests returned errors; the log showed ORA-12516 | The database had run out of processes, so the listener refused new connections |
| Stuck | Errors continued after the old process exited | The new pool had marked itself failed and did not recover by itself |
| Noise | The log grew by about 220 lines a second | Debug output printed stack traces for the connection failures |
| Recovery | Service back seven minutes after it began | Pool sizes cut, debug turned off, ORDS restarted cleanly |

Two things stand out. The outage began with a restart, not with load. And the step that ended it was a second restart, which is why "just restart it" is not a fix until you know why the first one failed.

## Cause 1: pools bigger than the database

In the usual dedicated-server setup, every connection in an ORDS pool is a server process on the database, and the database caps those processes with its `processes` parameter. Our pool was allowed 1000 connections. The database allowed 200 processes in total.

That 200 is not all available to ORDS. The database's own background processes count against it, and so does every other client: SQL*Plus sessions, schedulers, other applications. Check both the limit and how close you have come to it:

```
SELECT resource_name, current_utilization, max_utilization,
limit_value
FROM   v$resource_limit
WHERE  resource_name IN ('processes', 'sessions');
```

`max_utilization` is the high-water mark since startup. If it is already near `limit_value` on a normal day, there is no room for a restart.

### The arithmetic

The worst case is not one pool at its maximum. It is every pool that can be open at once, at its maximum, plus everything else:

```
needed = Σ over pools (MaxLimit × copies open at once) + background
+ other clients
```

"Copies open at once" is the term people forget. In normal running it is 1. During a restart that overlaps, it is 2, which is Cause 2.

With our original settings, a single overlapping restart needed room for 1000 + 100 connections against a limit of 200. Even `InitialLimit` alone, opened twice, used 200 slots before background processes were counted. The pool did not need heavy traffic to fail. It failed by starting.

### Rule of thumb

Set `MaxLimit` so that two copies of every pool, plus your observed peak from other clients, still fit under `processes` with room to spare. Set `InitialLimit` low, often 5 to 10, because a pool that grows on demand costs little, and one that opens 100 connections at startup makes every restart a stress test.

## Cause 2: the restart that overlapped

A restart is a stop followed by a start, and the gap between them is where this outage lived. If the new ORDS process starts before the old one has closed its connections, both pools are open at once. For a few seconds the database carries two copies of everything.

Overlap happens more often than you would expect:

- **Scripts that do not wait.** A stop that sends a signal and returns at once, followed by a start, gives no guarantee the old process has gone.
- **Two managers for one process.** ORDS started by hand, then "restarted" by systemd or a script that does not know about it, can leave the original running.

- **Slow shutdown.** Closing hundreds of connections takes time. A process that is still shutting down still holds its sessions.

### Waiting properly

Before starting, confirm the old Java process has exited, not just that the stop command returned:

```
# wait up to 60 s for the old ORDS process to exit
for i in $(seq 1 60); do
  pgrep -f 'ords.war' >/dev/null || break
  sleep 1
done
pgrep -f 'ords.war' && echo 'ORDS still running - not starting a
second copy' && exit 1
```

Adjust the `pgrep` pattern to match how your ORDS is launched, and make sure it cannot match a second, unrelated instance on the same host.

Under systemd, `systemctl restart` waits for the stop to finish before starting, but only for processes systemd started. If ORDS was ever started by hand, stop that copy first.

### After the start

A new ORDS process takes time to open its pools. Health checks that run in the first seconds will report failures that are really just startup. Give the instance a grace period, often 30 to 60 seconds, before you treat errors as real. Then check the pools directly instead of a web page: on 25.2 and later, `ords config --db-pool <name> verify` reports whether a pool is valid.

## Cause 3: debug output in production

`debug.printDebugToScreen` sends full error details, including Java stack traces, to the browser and the log. It is useful on a laptop. In production it did two kinds of damage during the outage.

**It buried the cause.** Each failed connection produced a stack trace dozens of lines long. At the peak the log grew by about 220 lines a second. The one line that mattered, ORA-12516, scrolled past in a wall of Java frames. Anyone tailing the log saw noise.

**It multiplied the load.** Writing hundreds of lines a second costs disk and CPU at the worst moment. Every tool that reads those logs (log shippers, monitoring, `tail -f` in three terminals) pays that cost again. A tool that reads every `*.log` file in a shared folder can count the same flood twice.

There is also a security cost: stack traces shown to the browser reveal class names, paths and SQL details to anyone who can trigger an error.

### Finding the cause in the noise

When the log floods, search for the database error, not the Java exception:

```
grep -o 'ORA-[0-9]\{5\}' ords.log | sort | uniq -c | sort -rn | head
```

One ORA code repeated thousands of times is a single problem, not thousands of problems. Grouping repeated errors this way is the quickest route from "everything is failing" to "the database is out of processes".

ORDS ships two settings profiles in `ords.war`: a production profile with debug off and a development profile with debug on. If you are not sure which values your instance started from, check before you need to know.

## The fix

Three settings changed, followed by one clean restart.

| Setting | Before | After | Why |
|---|---|---|---|
| `jdbc.InitialLimit` | 100 | 10 | Starting small makes restarts cheap |
| `jdbc.MaxLimit` | 1000 | 50 | Two copies plus other clients now fit under `processes` = 200 |

| Setting | Before | After | Why |
|---|---|---|---|
| `debug.printDebugToScreen` | true | false | Readable logs, no stack traces sent to browsers |

With the `ords` command line (the configuration folder is passed with `--config`):

```
ords --config /path/to/config config --db-pool default set
jdbc.InitialLimit 10
ords --config /path/to/config config --db-pool default set
jdbc.MaxLimit 50
ords --config /path/to/config config set --global
debug.printDebugToScreen false
```

Two habits made the change safe:

1. **Back up the settings file first.** Copy `settings.xml` with a timestamp in the name before any change, so undoing it is one command.
2. **Read the values back.** `ords config get` after each `set` confirms the key was spelled right. ORDS saves an unknown key with only a warning, so a typo looks like success.

Then stop ORDS, confirm the process has exited (Cause 2), and start it again. Watch the log for ORA codes during the first minute, not just for "started".

## Five safeguards

Any one of these would have shortened or prevented the outage. All five together make a restart boring, which is the goal.

1. **Size pools against the database, not the traffic.** Two copies of every pool at `MaxLimit`, plus other clients, must fit under `processes`. Check `v$resource_limit` before you choose numbers.
2. **Keep `InitialLimit` small.** 5 to 10 is enough for most instances; pools grow when they need to.
3. **Never run two copies.** Stop, wait for the process to exit, then start. One manager per ORDS process: if systemd runs it, nobody starts it by hand.

4. **Debug off in production.** Check `debug.printDebugToScreen` after every install and upgrade, because a development default can slip through.
5. **Watch the pools, not just the page.** After any restart, check pool validity (the Instance API `/status` reports valid and invalid pool counts, and `verify` does on 25.2+). A web page can answer while a pool is failed.

The common thread: each safeguard is something to check *before* a restart, because during the outage there was no time to work any of it out.

## Sidebar: how ORDSwatch watches this

This outage is part of why ORDSwatch exists. It flags oversized pools and debug settings left on in production *before* a restart. After a restart, it reports an instance as degraded when a pool is invalid, even if the web page still answers. And it groups a flood of repeated ORA errors into one incident, so the cause is the first thing you read rather than line 4,000 of a stack trace.

You can see it running against real ORDS instances, with no sign-up, at ordswatch.cinnamonsservices.com/live.