whogloo

STOP MOCKING EVERYTHING: TESTING OPENEDGE WITH REAL DATABASES

Disposable OpenEdge database containers for faster, production-accurate ABLUnit suites.

Session Agenda & Logistics

Thursday,November 5 • 11:45 PM – 12;45

01

The Dilemma

Why ABL resists mocking, and what shared dev databases cost us.

02

Tooling & Images

Building reference-database images for OpenEdge 12.8 and 13.0 with Docker and Podman.

03

Patterns & CI

Rollback-wrapped tests, reset strategies, schema evolution and pipeline integration.

01

THE DILEMMA

Why ABL resists mocking, and what shared dev databases cost us.

The TDD Dilemma in OpenEdge

Most OpenEdge shops test against one shared dev database — or not at all.



Shared Dev Database

Realistic schema, but state drifts, tests collide, and results aren't repeatable.



Mocks via Interfaces

Fast and isolated, but only possible where code is already OO and data flows through temp-tables or ProDataSets.

Why Mocking Fights ABL

In ABL, the database is part of the language.

Compile-Time Binding

FIND and FOR EACH buffers are resolved against the schema at compile time, and r-code carries table CRCs. There's no seam to mock through.

Rules Live in the Database

Triggers, unique indexes and index-driven ordering carry business rules that can only be verified against a real database.

The Bug a Mock Can Miss

A duplicate-key example shows why database behaviour matters.

Mocked Write

A test double accepts a second customer with an existing unique reference.

The application test passes.

Real Write

The database's unique index rejects the duplicate.

Check that the application handles the error and leaves no partial data.

Coverage Gap

The mock only checks behaviour it implements.

A database test exercises the actual constraint and the application's response.

Data-Driven Logic & False Confidence

When business logic depends on relational schema behaviour.

Hidden Database Coupling

ABL programs tightly integrate queries, temp-tables and database buffers.

The Mocking Trap

Mocking data access recreates the production schema inside test-double code, doubling maintenance.

Challenging Core Assumptions

Revisiting the old rule: "Databases should always be mocked away."

The Old Assumption

Databases are slow, shared, permanent state blocks requiring tedious manual resets.

The Modern Reality

Ephemeral containerised instances spin up clean per test run or test suite.

The Evolution of Modern TDD

Moving from hardware scarcity to software abundance.

Yesterday

One monolithic dev database shared across all developers and test runs.

Transition

Local script-based database creation utilities with fragile state cleanup.

Today

Containerised database lifecycles managed directly by the test pipeline.

THE CORE THESIS



Mocks make tests pass.
Real databases make systems work.

If your application is fundamentally data-driven, removing the database isn't isolation — it's avoidance. Modern TDD makes dependencies cheap enough that you don't have to fake them.

Shifting from isolation through avoidance to isolation through abundance.

02

TOOLING & IMAGES

Building reference-database images for OpenEdge 12.8 and 13.0 with Docker and Podman.

What Is Actually Isolated?

A private database is one part of an isolated test run.

Private Database

Give each run its own writable database copy.

Its records, sequence values and locks belong to that run.

External State

Shared report folders, services or queue names can still cause collisions.

Give each run its own resources where needed.

Defining Disposable Environments

What makes a testing database container truly disposable?

Ephemeral Lifespan

Created when the test run starts and destroyed on exit, with no persistent volumes.

Zero Side Effects

Every run starts from the same baked image, so parallel runs never share state.

Tooling: Docker & Podman in OpenEdge

Leveraging container engines for containerised OpenEdge database runtimes.

Container Engines

Docker and rootless Podman handle fast spin-up and teardown on workstations and CI agents.

Portability

Identical database environments across Linux x86-64 workstations, build servers and pipelines. Apple Silicon runs these images under emulation.

Why Podman & Rootless Containers?

Security and permission advantages in enterprise CI/CD environments.

Daemonless Architecture

No background daemon, and rootless mode runs without root privileges, reducing privilege-escalation exposure and simplifying security reviews.

Seamless Drop-In

`podman run` and `docker run` are command-line compatible, including `--secret` for license handling.

Image Strategy for OpenEdge Releases

Packaging OpenEdge binaries, schema and reference data into reusable images.

Base Images

One image per OpenEdge release and update (e.g. 12.8.x, 13.0.x), kept in a private registry. Licensed images never go to public registries.

Schema Layering

The database is created and the full .df loaded at build time. Boot only copies files and starts the broker.

Seed Data

Fixtures and sequence values loaded at build time for predictable starting states.

Multi-Version Compatibility Testing

Validating codebases against 12.2 LTS, 12.8 LTS and 13.0.

Matrix Builds

Pipelines parameterised by image tag to match client deployment targets.

Upgrade Safety

Compile per version and never share r-code across releases, so syntax and runtime differences surface before client upgrades.

Automated Schema & .df Loading

Building the reference database at image build time, not on boot.

Build Script

`prostrct create` + `procopy empty8`, then `prodict/load_df.p` in single-user batch mode (`_progres -b -db app -1`).

Fail Loudly

`load_df.p` reports errors in a `.e` file rather than throwing, so check for it and fail the build.

Seeding Reference Data & Fixtures

Populating deterministic datasets required for complex business logic.

Deterministic Seed Files

JSON or .d fixtures for lookup tables, currencies and system parameters, plus explicit sequence values, loaded at build time.

Isolation of Fixtures

Reference data stays immutable while test-specific transactional records are created and rolled back.

Choosing Useful Fixture Data

Small datasets with enough detail to exercise meaningful behaviour.

Reference Values

Include the lookup rows and system settings required for the scenario.

Keep values explicit and version them with the schema.

Edge Cases

Include missing optional values, boundary amounts and records with several dependent rows.

Every fixture should support a specific assertion.

Small Baseline

Use a compact everyday fixture for rapid tests.

Maintain a separate representative dataset for migration and performance checks.

Where do you start ?

Creating the base database to work from

Greenfield install

Create a new database using prodb, load the entire df, load seed data.

Testing this process gives us a “seed” database for the release

Migrations

Take version x of the database**, apply migrations in order to get to version y

Testing this process gives us a migrated database for the release

** preferably from an image with the older release

Create the images

Creating the base image to work from

Greenfield image

Create an image for this “fresh” database.

Migration images

Create an image for each migration path.

Always useful to create an image for a “larger” database in order to test scalability, timings, etc. A migration on a 100gb db may not run optimally ..

Keeping Reference Images Current

A reliable baseline needs a clear rebuild policy.

Schema & Fixtures

Rebuild when the .df or seed data changes.

A baseline should represent the schema and starting data for a specific release.

OpenEdge Runtime

Rebuild and retest when the OpenEdge release or update changes.

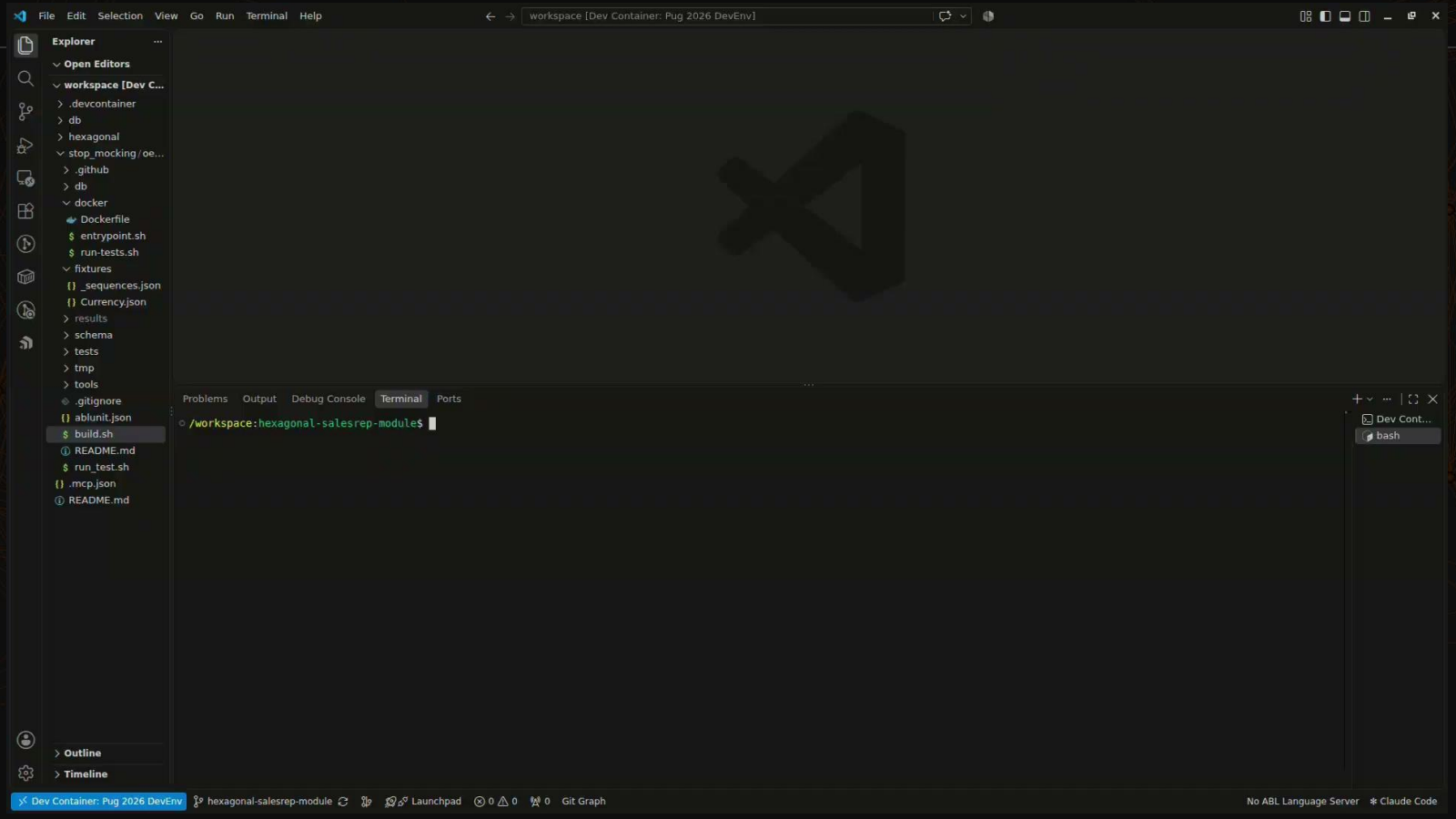
Verify the database loads and the intended runtime starts correctly.

Traceable Versions

Give each baseline a distinct image tag tied to its build inputs.

Record that tag in local runs and CI reports.

Creating the base image



From Image to Repeatable Baseline

The checks that connect the demonstration to everyday testing.

Built Database

Confirm the database contains the expected schema and seed data.

A successful image build must include checks on the resulting database.

Fresh Working Copy

Start a writable copy from the image. Check that it has the expected starting state before running tests.

Repeatable Start

Discard the working copy and create another.

Both should start with the same fixtures, regardless of changes made during the first run.

03

PATTERNS & CI

Rollback-wrapped tests, reset strategies, schema evolution and pipeline integration.

Data Setup & Teardown Patterns

Managing database state cleanly between ABLUnit test suites.

Ephemeral Resets

Recreate the container per run, or re-copy pristine database files between suites. Both take seconds.

ABLUnit Integration

`@BeforeEach` / `@AfterEach` around each test; `@BeforeAll` / `@AfterAll` once per class.

Proving Tests Are Repeatable

Repeatability should survive more than one successful run.

Repeat the Run

Run the suite twice from the same baseline.

Confirm the results match without manual cleanup or changes to the fixtures.

Change the Order

Run independent tests in a different order.

Failures can expose leftover records or hidden dependencies between tests.

Run in Parallel

Run two suites concurrently against separate database copies.

Confirm they produce the same results without colliding on external resources.

ABLUnit Annotations in Practice (12.8+)

Structuring test lifecycle methods inside ABL classes.

@BeforeEach / @AfterEach

Run around every test. The legacy @Setup and @TearDown still work but are deprecated.

@BeforeAll / @AfterAll

Run once per class. Watch out: the legacy @Before and @After meant *this*, not per-test.

Rollback Wrapping

ABL has no nested transactions. An outer `DO TRANSACTION … UNDO` turns the tested code's transactions into subtransactions.

Rollback Caveats

What an outer UNDO won't reverse.

Not Undone

Sequence NEXT-VALUE, NO-UNDO variables and temp-tables, and anything committed through PASOE or AppServer calls.

Block Scope

A transaction can't start in @BeforeEach and end in @AfterEach. The wrapper belongs inside the test method.

Rollback Tests and Commit Tests

Choosing tests that match the behaviour you need to verify.

Rollback Wrapper

Check changes inside the transaction, then UNDO for cleanup.

The wrapper changes transaction boundaries, so this test covers behaviour before commit.

Committed State

Run the operation without an outer test transaction.

After it completes, use a fresh session to check what persisted, then reset the database.

The Reset Ladder

Choosing the cheapest reset that keeps tests honest.

ms

Per Test

Rollback-wrapped transactions.

secs

Per Suite

Shut down the database, re-copy the pristine files, restart.

secs

Per Run

A fresh container from the image.

How long does this take

Because every second matters. Or does it ?

ms

Small db

Fixture data

secs

Medium db

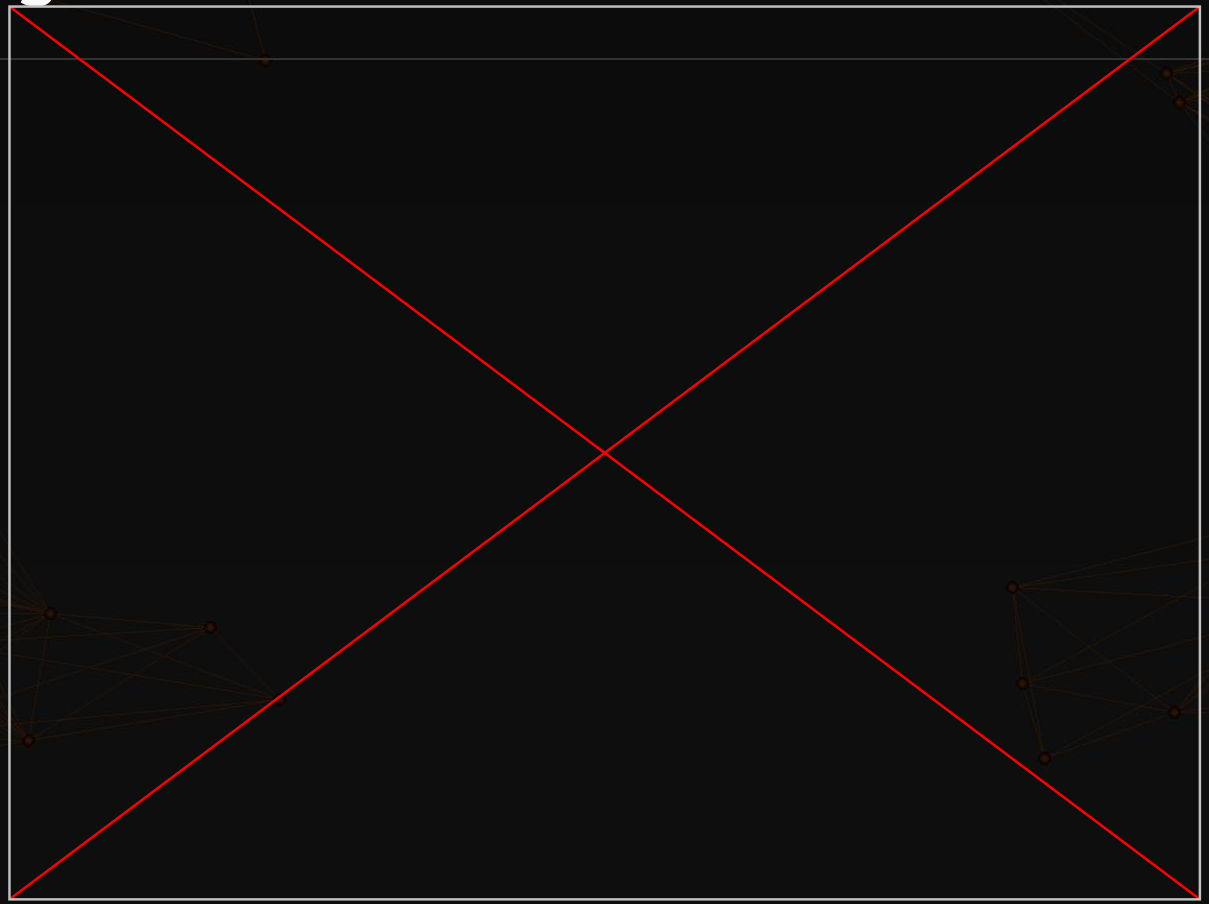
Fixtures + basic samples

secs

Large db

UAT based on customer db

Running tests



Maintaining Fast Feedback Cycles

Balancing high realism with developer agility and execution speed.

Container Reuse

Keep one container running across local iterations. Safe only with rollback-wrapped tests or resets between suites.

Parallel Suites

Give each runner its own container. Publish `-S` plus the `-minport/-maxport` range, or run tests inside the database container.

Targeted Execution

Run unit and integration suites independently during local loops.

Testing Contention Between Sessions

Locking behaviour needs two sessions sharing one database.

Two Sessions

Coordinate two AVMs on one multi-user database.

Session A holds EXCLUSIVE-LOCK while session B attempts to update the same record.

Contention Result

Use NO-WAIT for a controlled attempt.

Assert the expected lock conflict and confirm the application handles it correctly.

Released Lock

After A ends its transaction and releases the record, B retries.

Confirm the update succeeds and the final values are correct.

Performance Tuning in Containers

Startup parameters for fast, disposable test databases.

No-Integrity Mode (-i)

Skips crash-recovery overhead. Acceptable only because the database is disposable.

Right-Sized -B

A buffer pool large enough to cache a small test database. Checkpoints and BI writes still happen.

tmpfs Storage

Copy the baked database into a tmpfs at startup. Mounting tmpfs over the baked path hides the files.

Schema Evolution Between Releases

Keeping test databases aligned with every release.

Source of Truth

The full .df is versioned at every release tag.

Generated Deltas

`prodict/dump_inc.p` compares old and new. A renames file stops renames from becoming DROP + ADD.

Drift Check in CI

N-1 image plus deltas must equal N built from the full .df.

Migration Checks Beyond the .df

A matching schema is only part of a successful upgrade.

Schema Structure

Compare the migrated schema with the full release .df.

This checks structure but does not establish whether existing business data survived.

Existing Data

Check representative old records and their relationships after upgrade. Assert totals, status values and required reference data remain correct.

Sequence State

Check migrated sequence settings against retained record keys.

Creating a new record should produce a valid key without collisions.

Traditional Mocking vs. Containerized Testing

A balanced perspective on where each approach succeeds.

Mock-Heavy Testing

Strengths: Fast, no environment dependencies, wherever OO seams exist.

Weaknesses: False confidence, fragile test doubles, and much of ABL can't be mocked at all.

Containerized Integration

Strengths: Production parity; verifies triggers and indexes.

Weaknesses: Image management, licensing and CI tooling overhead.

Choosing the Right Tool for the Job

A hybrid pyramid model for enterprise OpenEdge applications.

Pure Unit Tests

Use mocks for external web services, UI controllers and isolated algorithmic utility classes.

Data Integration Tests

Use disposable database containers for Business Logic Modules, triggers and complex database transactions.

Pipeline Integration: Jenkins & GitHub Actions

Running containerised test suites reliably inside build pipelines.

Database Containers

Service containers or `docker run` in GitHub Actions; `docker.image().withRun` or docker compose in Jenkins.

Automated Reporting

ABLUnit writes a JUnit-style `results.xml` for pipeline dashboards.

How CI Knows the Tests Ran

A green build needs evidence of successful test execution.

Expected Tests

Check that the expected suites ran and produced a fresh report.

An empty or missing result should fail the job.

Failing Tests

Preserve process errors and inspect test failures.

Deliberately break an assertion to prove that CI reports a failed build.

Failure Diagnostics

Keep ABLUnit reports and runner logs as artifacts.

Include the image tag and schema version so failures can be reproduced.

Overcoming Adoption Hurdles

Practical advice for introducing container testing to established OpenEdge teams.

Incremental Migration

Start with critical database integration suites before touching legacy unit tests.

Developer Training

Give developers simple wrapper scripts to run containers locally.

License Management

Containers still need your OpenEdge license. Confirm dev/test and CI terms with Progress, mount `progress.cfg` at runtime, and never bake it into images.

A First-Week Adoption Plan

Starting small with existing code and a measurable result.

One Useful Suite

Choose a database operation with real business risk.

Run an existing test suite against it using one disposable baseline.

One Command

Provide the same wrapper for local development and CI.

Developers should be able to run the suite without manual database setup.

Evidence to Expand

Record total runtime and test repeatability.

Add more suites once the first one gives useful, dependable feedback.

Key Session Takeaways

Core principles to bring back to your development team.

Embrace Realism

Stop masking relational behaviour behind fragile mocks.

Leverage Containers

Disposable, identical databases from one baked image.

Version Your Schema

Full .df plus deltas plus a drift check.



whogloo

Modern TDD isn't about pretending dependencies don't exist.

It's about making them cheap enough that you don't have to avoid them.

Thank you | OpenEdge ABLUnit session