Pluggable Databases for DevOps

Making Individual Developer Databases a Reality

Duane Dieterich, DMSTEX
Amber Yancey, ETCC
IOUG Collaborate 2019

Abstract

In a DevOps world, database development presents some unique challenges. Adequate source control and modern DevOps practices such as continuous-integration make necessary the need for individual development databases, which is a big change from the shared development database. Determining how to build and test these databases can be a complex challenge. How to manage the possibility of databases and database software scattered across a large development environment can be daunting.

In this presentation, we identify our solution to some of these issues. We modified our existing continuous-integration process for java to make possible the creation of stand-alone version-controlled database builds. We utilized several key technologies to make our implementation as practical as possible, including Docker for the backbone installation handler of each component in our build server system. Jenkins was used as the actual build server/UI, which we embedded with SQLPlus, Dockerized Oracle was the basis for our SQL server, Nexus served as our build repository, and Maven was used to assist with management of version control. All of this was made possible because the Oracle pluggable database is manageable in size and quick to load. We will also uncover some problems we encountered with this implementation.

Presentation Attributes

Session Type	Case Study
Sub-Category	Dev/Ops (CiCD, Docker)
Learning Objectives	Identify problem with development databases.
	Show an implementation using a build server and pluggable databases.
	Identify lessons learned from implementation.
Content Level	Mid-level. Discuss what users need to consider.
Product Lines	Oracle Database, Oracle Database Multitenant, Docker
Industry	Applicable to All
Company Type	User (Oracle Database User)

Table of Contents

Executive Summary

Database development presents some unique issues for development environments. The development problems related to shared databases are a result of conflict between different versions of database objects and competing requirements from developers. The problems related to individual development databases are caused by the proliferation of databases in the development environment.

An implementation can use script files as the source code for database schema and data. These script files can be stored in Git. A build server can be used to convert any version of database source code into a working database. These databases can be repeatedly loaded on, and removed from, developer workstations. Oracle's pluggable database is the fastest form of database storage for this effort. A branching model and matching PDBs with source code versions (branches) are crucial.

Key lessons learned show that the unique problems of building and testing databases on a build server can be resolved. There are several problems that can be resolved when loading PDBs on a developer workstation. Specific challenges for Windows7 and Docker on the developer workstation can be resolved.

The Problem with Development Databases

Shared Development Databases

In many environments, a development database is shared by multiple developers. One problem with a shared development database is mixed versions of database objects. These mixed versions cause problems when there are dependencies between the objects. A new version of one object can cause another object to become invalid. A different version of one object can cause another object to stop working correctly. These different versions and associated dependencies cause frustration between developers and inhibit productivity.

Coordinating multiple changes to the same database object can also cause problems in a shared development database. If these changes need to remain separated, the developers involved with these changes must coordinate which set of changes are in the database at any given time. Determining the need to coordinate with another developer can also be difficult. Time spent searching for the source of changes to a database object and coordinating with other developers can cause frustration and obstruct progress.

Because a database persists data, a special kind of cross-test data interference can occur. This happens when multiple developers expect to modify the same data, or data modified by one developer changes the test results for another developer. Determining the need to coordinate with another developer can also be difficult. Time spent searching for the source of changes to a database object and coordinating with other developers can cause frustration and inhibit productivity.

Catastrophic test failures can halt all development on a shared database. These failures require database restoration or rebuild before shared development work can continue. Unfamiliar data in a different or rebuilt database can causes delays in development lead times. This results in frustration and inhibited productivity.

If a shared database is taken offline for maintenance, shared development halts. Problems similar to catastrophic test failures cause the same problems mentioned above. These are a few of the problems associated with shared database development.

Individual Development Databases

Separate databases for each database developer can also be a source of difficulty. One problem is keeping the objects current in the individual databases. Without some mechanism to keep the database objects updated, they grow stale and unusable. Further difficulty occurs when an independent sets of changes are being worked at the same time. Different versions of different objects are needed, depending on which set of changes is being worked.

Continuously rebuilding and/or refreshing the individual database is also a problem. The rebuild/refresh can tie-up a development workstation, preventing it from being used for other

work. Management of database object refreshing requires tight coordination between source control and the database. Existing data in the database presents a problem when refreshing tables or types.

Distributed updates of Oracle database software (Oracle patches) can be a problem with individual databases. Ensuring the correct updates have been applied to every developer's database can be complex. Update failures are problematic when development schedules don't include time to work through the problem.

Whether the database is shared or not, developers spend valuable time establishing test data. If individual databases are fully refreshed, this can eliminate the carefully prepared test data. Having enough data in a refreshed database can also be problematic.

If a distributed source control system like Git is used, there is an added complexity in coordinating database changes between the local Git repository and one or more remote Git repositories. Different versions of database objects will exist in various branches within each of those repositories causing confusion and difficulty in keeping individual databases up to date. These are some of the problems that can occur with individual development databases.

Building the PDB

Presented here is an implementation for individual development databases. This implementation solves the problems with individual development databases that are listed above. The solution is described in 5 major areas.

Database Source Code Capture

The starting point is a source code capture utility. This utility captures the schema (DDL) and data (DML) required to recreate an existing database. This utility is different than cloning because text files are created instead of copying database files. The utility is configured to create installation scripts for basic schema and configuration data. It creates installation scripts for various environments like development, test, or prod. It captures additional schema and data for other needs like quality assurance database objects and test data.

The source code capture utility creates installation scripts that run against an empty database. The installation sequence starts with creating tablespaces, roles, and schema owners. Installation scripts are grouped into installation types. Installation types are needed to populate databases differently based on how the database is used. For instance, populating a database to be used for development may need QA database objects and test data. However, a production database does not.

There is a single installation script hierarchy for each installation type. Within the installation script hierarchy, different objects for multiple schema owners are populated at the same time. This allows an error free installation sequence.

The installation scripts only use SQL*PLus and SQL*Loader to populate a database. Licenses to use SQL*PLus and SQL*Loader are included with a (non-cloud) Oracle database license. The SQL*PLus and SQL*Loader utilities are included in Oracle database software installations. Instant client versions of SQL*PLus and SQL*Loader are available for simplified installation on a remote database server.

The installation scripts created by the source code capture utility is the database source code for the system begin developed. Adding this database source code to a version control system like Git allows database development to occur just like non-database development. Different versions of the database source code can be retrieved for building version specific databases. Because the installation scripts are text files, different versions of the database source code can be easily compared. Different versions of the database source code can be merged to create installation scripts with combined capabilities.

Version Control

Git is used for the version control system. GitLab is the server used to manage the central repository. The GitLab web interface is used on a daily basis. Almost every task or defect is

tracked in the GitLab issue tracker. Project development documentation (mostly procedures) is placed in the GitLab Wiki.

Each workstation clones the central repository using the Git Extensions client. Git Extensions is used to checkout and pull branches that are not associate with a PDB. Git Extensions is also used to stage, commit, and push local changes to the remote repository.

A simple webhook is used in GitLab to kick-off builds on the Jenkins build server when a developer pushes their code. In addition, Git hooks (triggers) are also used to assist with build management and version control management. These are the main hooks:

- **post-receive.modify-new-branch-poms** - Identifies branches as either "development", "feature", or "release" branches.. This trigger updates the version in the branch's POM.XML file(s) and pushes the changes to GitLab.
- **pre-receive.01.lock-push-during-build** - Grabs the incoming branch name, checks to see if a build is currently running for that project and branch name, and fails for all users except those in the exception list.
- **pre-receive.02.branch-name-check** - Identifies a new "development", "feature", or "release" branch and passes or fails branch creation based on naming convention checks.
- **pre-receive.03.pom-admin-check** - Allows certain users to push changes to the POM.XML and Jenkinsfile files. This also prevents users from pushing Git "notes".

Each PDB includes the version from the POM.XML file. The database installation scripts include a mechanism for loading this version in the database. Each installation type is loaded with the version from the POM file.

Pluggable Database Build Environment

The build environment started with Jenkins, Git, Maven, and Nexus from previous projects. Oracle database and Oracle instant clients were then added to the build environment. The ability to create a new PDB in the build environment was developed. The database installation scripts from Git were used to install schema and data. The build environment was enhanced to run the unit testing (wtPLSQL), unplug the PDB, zip the PDB files, and copy the zipped PDB files to a Nexus server. JUnit XML reports were created for installation log files, valid database object reports, and unit test results.

Docker also presented several advantages. Docker containers for Jenkins build servers and Maven servers were already in use. Oracle also provides a Dockerfile to create an Oracle database in a Docker container. Additionally, utilization of Docker containers allowed the use of different versions of the Oracle database as required for different projects.

The build logs are stored inside the Jenkins container in a volume that is mounted to the build server. Access is available from the build server. These build logs include the logs from the installation scripts, logs from SQL*Loader, and the JUnit XML files. Additional status

information like number and size of PDB files are also included in these logs.

PDB Development Process

Branching Model

Previous experience with CVS revealed that too many branches can ruin a project. In order to keep branching under control in Git, a branching model was created for development and release cycles. In the model, development starts with a development branch that is set to the default branch. No changes are made in this branch. Developers create new branches from the default branch for each feature/defect being worked.

One or more feature/defect branches are merged into one or more release candidate branches. This gives the change control board some flexibility in deciding which set of features/defects will go into the next release. When a release candidate branch is ready, a set of diff scripts is created to upgrade and downgrade the database. After a release candidate branch is selected, it is released and the next development branch is created and set to the default branch.

Usually a separate Git branch is used for each defect or enhancement. An exception would be a set of defects or enhancements that are tightly coupled. If multiple developers need to work on the same branch, a single database on a developer workstation can be shared by the database developers.

Workstation Components

Administration of individual development databases was a top concern. It is estimated that approximately 140 developers would need development databases. It was determined that each developer would have 3 Docker containers with databases. Distributing Docker containers would provide an easy route for reconfiguration and patching databases among the developers.

In the early stages, different database loading methods were performance tested. PDB loading was the fastest mechanism. Restoring cold backups was slower because the file handling was the slowest part of the load and the cold backups were larger than the PDBs. Building the database with scripts was the slowest. Export/Import were not considered because of the complexity.

PDBs were very attractive for portability. Oracle's documentation identifies the portability of these database files, including the conversion between big endian and little endian. We settled on an environment that used a CentOS Linux server for PDB builds and Window for development workstations. This portability works as advertised.

With one or more PDBs loaded, development and testing can occur in these databases on the developer workstation. After development and test is complete, a set of scripts must be

updated with the changes from the database. The same utility used to capture the schema (DDL) and data (DML) scripts from the original database is also used here. Script files are deleted and recreated with all database changes using the utility. The staging operation in Git Extensions allows review of all script changes before committing the changes.

Loading as many as 3 different PDBs on a developer workstation allows a developer to work 3 different Git branches at the same time. Initially, the Git repository was cloned 3 times, one for each PDB. However, Git working trees are much faster. Git working trees allow the checkout of multiple branches from the same local repository. (It is important to note that Git documentation sometimes uses working trees, workspaces, and work-trees interchangeably.) During normal cloning, a workspace is loaded with a set of script files. The Git working tree command creates additional sets of scripts files, each set for a different Git branch. With 3 PDBs loaded and 3 matching working trees, a developer can work multiple branches simultaneously.

Workstation Automation Scripts

In order to facilitate movement of PDBs on and off the developer's workstation, a set of scripts was developed to accomplish 3 groups of functions. The first group of functions is centered on loading a PDB on the workstation. Loading a PDB involves the following functions:

- Input "slot" from user
- Input Git branch from user
- Identify matching PDB
- Identify working tree and PDB file locations
- Confirm available PDB file space
- Confirm unique branch in working trees
- Download, unzip, and plug-in PDB
- Fetch and checkout remote Git branch

The second group of functions involves removing a PDB from the workstation. Removing PDBs involves the following functions:

- Input "slot" from user
- Unplug the PDB
- Backfill the working tree with a "No PDB" branch

The third group of functions lists the status of the working trees. This group of functions is important to understand which branches/PDBs are loaded and the status of the working trees. Listing the status of working trees involves the following functions:

- Identify all working trees
- Report working tree location
- Report number of files in the working tree location
- Report number of files changed, but not staged
- Report number of files staged, but not committed
- Report number of commits that have not been pushed
- Report branch name

Lessons Learned

Building a Pluggable Database

As PDBs were built, log and trace files were created by the Oracle database. At some point, the build server stopped creating PDBs because there was no more space in the Docker container. We automated a task to periodically purge the oldest log and trace files from the Docker container.

An unplugged PDB is basically a compressed archive of the database files (and XML file) needed to plug-in that PDB. Oracle 12.2 includes an SQL command that will create this compressed archive. Since Oracle 12.1 does not, our PDB build scripts had to unplug and "zip" the needed PDB files. On the developer workstation, our PDB load scripts had to "unzip" the files to the proper location, then plug-in the PDB.

After a PDB build, a JUnit XML report is created showing passed tests for valid objects, failed tests for invalid objects. PL/SQL APIs and tables from the other databases are required to successfully compile all database objects. The database schema includes database links to these other databases. During the database build, a set of "stub" schemas and objects are populated. These stub objects emulate the API and table structures needed from the other databases. "Loopback" database links are created to access these stubbed database objects.

Oracle's "create pluggable database" command uses a seed PDB by default. This seed database is included with an Oracle database software installation. Creating a custom seed PDB with as much static content as possible reduces build time. For instance, the APEX_EPG_CONFIG script loads a large number of binary files into the PDB. Having these images already loaded in a custom seed PDB reduces build time on the build server.

Loading a Pluggable Database

Loading the correct PDB for a Git branch is crucial. Each branch may have zero or more PDBs. Manually finding the correct PDB is prone to errors. A script was created to load the correct PDB for a branch. The script does a Git checkout of the remote branch and pulls any differences. The script checks the Nexus repository based on the commit hash sequence of the branch. The commit hash does not indicate the type of branch. So, the script uses Nexus queries that include other patterns/masks to limit the search results. Following is the order for the PDB search in Nexus.

1. Hash for Given Branch
2. Hash for Development Branch
3. Hash for Previous Release Branch
4. Hashes for Previous Development Branch

Several PL/SQL programs in the PDB use UTL_FILE to read and write files. The location of these files is determined by Oracle directories. The path for these directories is different based

on the “slot” for the PDB. For instance, files may need to be opened for “/workspace/PDB2” instead of “/workspace/PDB1”. The PDB load script updates the Oracle directories with the correct location for UTL_FILE.

Oracle Enterprise Manager Express and Oracle Application Express are available from each PDB using a URL. However, they cannot share the same network port. The PDB load script modifies the port used by the embedded PL/SQL gateway during the PDB load.

The PDB built on the Linux server would not run Java methods on the Windows workstation. The JDK in the PDB had to be updated. Oracle provides the “update_javavm_db.sql” script to correct this problem. It must be run every time a PDB is loaded.

Developer Workstations

Running Docker on Windows 7 requires Virtualbox. With the three levels of operating system hierarchy, any files shared between the Docker container and Windows had to be coordinated through Virtualbox. Because Virtualbox requires a complete operating system within, it was quite large. The startup sequence of Virtualbox, Docker, and the Oracle database was slow and made for a large Vitualbox VM folder.

Amazon workspaces won’t run a Docker container under Windows (or any other operating system). Support for Windows desktop workstations and Windows virtual workstations on AWS did not allow the same deployment scheme. Docker was dropped and the Oracle database was installed directly in Windows to simplify initial project startup.

Oracle 12.1 on Windows won’t release all files when the PDB is unplugged. This was initially handled by restarting the container database every time a PDB was unplugged on the Windows workstation. However, this was changed to prevent interruption of work on other PDBs.

Each time a Windows workstation is restarted, all PDBs remain closed after container database startup. The starting of the PDBs remains manual to maintain experience with Oracle Enterprise Manager Express. Consistent use of OEM Express builds developer confidence to administrate their own databases.

Initially, worktrees were set back to an old master when PDBs were removed. However, there was a long delay during branch checkout when a PDB was loaded. Several "No PDB" branches were created and used to “idle” worktrees. The special branches need to be updated periodically to keep the number of file changes small when PDBs are loaded or removed.