Implementing DevOps principles with Oracle Database

October, 2023, Version 1.1
Copyright © 2023, Oracle and/or its affiliates
Public

ORACLE

Purpose statement

This document provides an overview of modern application development principles, in the context of Oracle Database. It is intended solely to help you assess the business benefits of changing your application development workflow to deploy code changes in an automated fashion. Due to the almost infinite number of permutations possible in development practices, this tech brief can merely list suggestions and best practices. Concrete adoption always depends on your requirements, skills, processes, etc.

The intended audience comprises developers, architects, and managers leading those teams that utilize Oracle Database as part of the database estate. Whilst every effort has been made to make this document as accessible as possible, a basic understanding of the Oracle Database is needed to take full potential of this tech brief.

Copyright © 2023, Oracle and/or its affiliates / Public

ORACLE

Table of contents

Introduction 6

- Towards a Modern Software Development Workflow 6
- Implementing Continuous Integration 7
- Continuous Delivery and Deployment 9
- Attributes of Continuous Integration and Delivery 9
  - Version Control 9
  - Fast Feedback Loops 10
  - Automated Testing 10
  - Small and Frequent Changes 10
  - Automated Deployment 10
- Summary 11

Using Version Control Systems 12

- No CI/CD without the use of a version control system! 12
- Getting the team's buy-in to using a version control system 12
- Introduction to Git 12
  - Git Terminology 13
  - Branching and Merging 13
  - Pull and Merge Requests 13
  - Forking a Project 14
  - Your Database Project's Git Repository 14
- Release Management 16
- Integrating with the Continuous Integration Pipeline 16
- Summary 16

Ensuring Repeatable, Idempotent Schema Migrations 17

- Benefits of using Dedicated Schema Migration Tools 17
- Using SQLcl and Liquibase to deploy Schema Migrations 18
  - Liquibase Terminology and Basic Concepts 18
  - Practical Aspects of Creating the Database Changelog 19
  - Formatting your changelog 19
  - Deploying the Changelog 20
  - Checking the status of your deployments 21
  - Ensuring Pipeline Failure 23
- Summary 24

Efficient and Quick Provisioning of Test Databases 25

- Autonomous Database 25
- Using Container Images 26
  - Using container images with Podman or Docker 27
  - Using Container Images with Kubernetes 28
- Using Container Databases 29
  - Creating a new, empty Pluggable Database 29

Copyright © 2023, Oracle and/or its affiliates / Public

Cloning an existing Pluggable Database 29
Automating Pluggable Database Lifecycle Management 30
Using Recovery Manager's duplicate command 30
Using block-device Cloning Technology 31
Using Copy-on-Write Technology 31
Using Schema Provisioning 31
Summary 31
Writing effective CI/CD Pipelines 33
Introduction to CI/CD Pipelines 33
CI Pipeline Stages 34
CI Pipeline Jobs 34
Ensuring Code Quality 35
Linting 35
Unit Testing 35
Performance Testing 38
Deployment 38
Updating the CI database 39
Summary 39
Performing Schema Changes Online 40
Deploying to Production with Confidence 40
Avoiding outages, however brief they might be 40
Online operations 40
Creating Indexes Online 40
Introducing partitioning to an existing, non-partitioned table 41
Compressing a segment online 41
Adding Columns to Tables 42
Using Online Table Redefinition to Change Table Structures Online 42
Next-level Availability: Edition-Based Redefinition 45
EBR Concepts 45
Adoption Levels 46
Potential Workflows 46
Summary 47

Copyright © 2023, Oracle and/or its affiliates / Public

List of figures

Figure 1. Greatly simplified view of a CI/CD setup 8

Figure 2. Advantages of using Liquibase to deploy schema changes 18

Figure 3. Liquibase changelog, changeset and change type explained 19

Figure 4. Screenshot showing Database Action’s Liquibase deployment screen 22

Figure 5. Screenshot showing Database container images on Oracle’s container registry 27

Figure 6. Using Container Databases to quickly provision test environments 29

Figure 7. Example of a successful pipeline execution in GitLab 33

Figure 8. Example of a successful merge pipeline execution in GitLab 34

Copyright © 2023, Oracle and/or its affiliates / Public

Introduction

Anyone in the market of providing software services to its customers has felt the pressure to innovate and – consequently – improve the software development process. The competition is never asleep and most companies in this highly competitive field simply cannot afford not to move at the same pace.

Releasing new features frequently can be a challenge if lead times are (too) long. In many cases, this is down to a low cadence of software release cycles. When it was acceptable to release new functionality once every quarter, the release process very often was not automated, resulting in Database Administrators (DBAs) applying changes to production during the night or on weekends.

Apart from the potential problems on release day, a lot of potentially time-consuming and error-prone human intervention was necessary to resolve typical problems associated with a new release such as:

- Merging many long-running feature branches into the release (or main) branch.
- Collecting all changes required for a software release.
- Creating a software release, especially when it involves a database.
- Performing meaningful integration/acceptance/performance tests prior to go-live.

Whilst developers have been able to take huge strides in automating and bundling software releases for e.g. frontend applications for many years now, database applications have not enjoyed the same level of attention and automation.

Many developers have tried to compensate for the perceived shortcomings with their backend data store by moving functionality typically belonging into databases into their applications. This is not a satisfactory solution since it tends to mask or postpone the problems inevitably following such as:

- Issues with data governance.
- Data quality can be negatively impacted if the application isn't used to enter data.
- Scalability might suffer as load increases, especially in the case of very chatty applications.
- Maintenance of application functionality that is provided by the database.
- Duplication of database-provided, redeveloped functionality across different applications.

Automation of database application deployments is still in its infancy in many software projects. Too often changes to the database, going forward referred to as *database releases*, are created outside a version control repository. In the worst case, such database releases may consist of several scripts combined in a ZIP file shared via email, executed manually on a database during a downtime window.

As you can imagine this process – although it might be proven over time – does not scale well with the requirement to release features more often. There is also a high risk of deploying changes without any traceability of *which* changes have occurred or *when*.

If your application is subject to fast release cycles there is no practical alternative to automation. As a welcome side effect automation can also lower the cognitive burden on the database professional handling the database release.

Towards a Modern Software Development Workflow

Rather than following a workflow featuring many long-lived branches with the associated problems during the merge phase, a different approach might be necessary if release frequencies are to improve.

Ground-breaking research has been published by Nicole Forsgren at al. in their well-known "Accelerate-the science of lean devops and devops building and scaling high performance technology organizations" book in 2018. Since then, the State of DevOps report has been published annually containing findings from thousands of individuals.

Copyright © 2023, Oracle and/or its affiliates / Public

One of the key messages is that high-performing teams **deploy code more frequently**, **quicker**, and with **fewer errors** than their peers. Key metrics to keep in mind include the following:

- Lead Time for Changes
- Mean Time to Recovery
- Deployment Frequency
- Change Failure Rate
- Reliability

This is remarkable because for a long time the industry believed that frequent releases must certainly come at the expense of quality. This has proven to be not the case if done right. The following sections explain tools and methods on how to improve code quality.

A key component to successfully deploy code quicker has been identified by the aforementioned book as **Continuous Integration/Continuous Deployment** (**CI/CD**).

Continuous Integration is a process where code changes are tested against the existing code base with every commit in a version control system (VCS) like Git.

Automating the Continuous Integration of your software is based on the following pre-requisites:

- All code must be subjected to version control using tools like Git.
- The focus must be on small, incremental changes ("work in small batches").
- Perform frequent integration runs to your main branch to ensure that your changes can merge successfully rather than letting them sit in a feature branch for days or weeks.
- Code coverage, formatting and syntax checking must be automated after the team(s) have agreed on standards.
- Tests are an essential part of the software project and must pass prior to a deployment to higher-tier environments. No code should be added without a corresponding test ("**Test-Driven Development**").

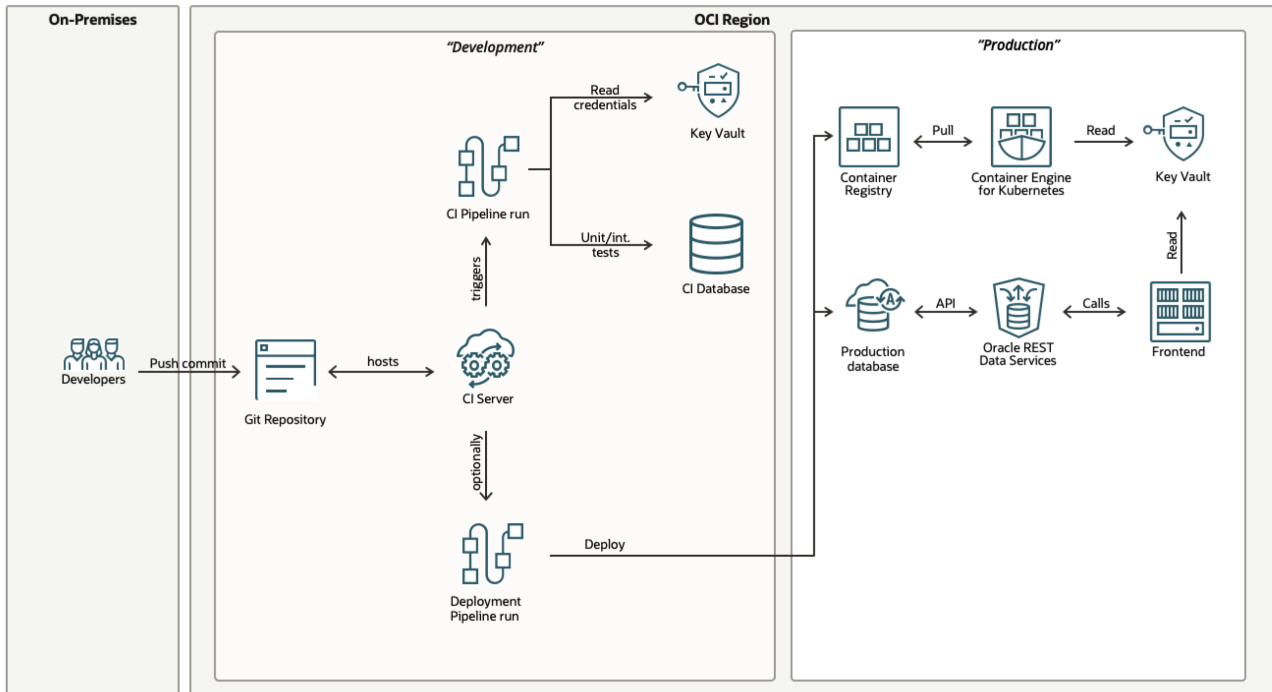
It is interesting to note that some of these requirements are technical, whilst others are not. Adopting a culture where everything is done via a version control system (VCS) like Git might require a significant effort by the change management team. Ultimately it is worth the effort to convince everyone on the team that development based on a VCS is the best way moving forward.

Implementing Continuous Integration

There are many tools available to support teams with Continuous Integration (CI). The central piece of the architecture is undoubtedly the CI server. It coordinates the execution of tasks, shows the results on a dashboard and often provides a method to manage and track software issues. When running tests, the CI server usually enlists the help of auxiliary infrastructure like a development environment for database changes, credential helpers like vaults and keystores as well as Kubernetes clusters or Container Engine runtimes as target platforms. The following figure demonstrates a simplified setup in **Oracle Cloud Infrastructure** (**OCI**):

Copyright © 2023, Oracle and/or its affiliates / Public

Figure 1. Greatly simplified view of a CI/CD setup



A central element of the CI server is the CI pipeline. In software projects, pipelines often consist of multiple stages like "linting", "build", "test", and "deploy to software repository". Each stage is further subdivided into tasks.

The "linting" stage typically consists of linting, code coverage, and security vulnerability checks. The "build" phase typically encompasses the building of an artifact/container image and so on. In the context of database applications, the build phase often comprises the creation of a test environment, automatic deployment of schema changes, and the execution of unit and/or integration tests.

A new commit to a source code repository typically triggers the execution of a pipeline. One of the central paradigms of CI is the rule to "always keep the pipeline green". This refers to the pipeline's status as indicated by a "traffic light" symbol on the dashboard: if the build fails for any reason (red status) developers must scramble to fix the error so as not to interrupt the workflow for others.

CI pipelines are often defined in YAML (GitHub, GitLab, for example) or other domain-specific languages. It is recommended that the CI server you choose for your project(s) allows you to store the (non-binary) representation of your pipeline along the source code. This goes back to the principle that everything in your project is version controlled.

The following snippet is an excerpt of a CI pipeline as used in GitLab:

```
# ------------------------------------------------------------ global variables
variables:
  CLONE_PDB_NAME: "t$CI_COMMIT_SHORT_SHA"
  SRC_PDB_NAME:   "SRCPDB"
  TAG_NAME:       "t$CI_COMMIT_SHORT_SHA"

# ------------------------------------------------------------ stage definition
stages:
- linting
- build
- test
- update-ci-db
- deploy
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
# ----------------------------------------------------------- linting

# use the docker executor to get the latest node image and install
# eslint. All JavaScript source files that are part of the release
# will be linted as per .eslintrc

lint-mle-modules:
  image: node
  stage: linting
  script:
  - npm install eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin
  - npx eslint r*/migrations/*.ts
  tags:
  - docker

# … additional steps
```

Continuous Delivery and Deployment

Many software projects take Continuous Integration a step further by delivering the fully tested, deployment-ready artifact to another tier like User Acceptance Testing (UAT) or even production, after all tests in the integration pipeline have passed. A pipeline is in use again, this time however it's referred to as a deployment pipeline.

Very often the line between **Continuous Delivery and Deployment (CD)** is drawn based on the degree of automation. With **Continuous Deployment** it is assumed that a build goes straight to production provided it passes all the checks and tests. **Continuous Delivery** is very similar, except that the actual deployment is triggered by an operator rather than the pipeline. Without Continuous Delivery one cannot have Continuous Deployment.

It should be noted that Continuous Deployment is difficult to implement, even for stateless applications. It requires a lot of commitment, resources, training, and a robust testing framework to be able to push each change to production with confidence.

Attributes of Continuous Integration and Delivery

A few key principles of CI/CD merit a closer investigation in the context of this paper.

Version Control

As you read earlier, it's impossible to implement automation of testing and deployment of applications without version control. Apart from being a requirement for all kinds of automation, there are numerous other advantages to using a version control system such as:

- Enabling a distributed workflow with multiple developers.
- Powerful conflict resolution tools.
- Being able to trace application changes back in time.
- Comparing code before/after a certain deployment/milestone/commit.
- Code is automatically backed up when using a central code repository.

The VCS system is critical to the infrastructure and must be set up in a fault-tolerant way. Should access to the VCS be down, critical parts of the automation pipeline will fail, preventing any changes from being made to production.

Copyright © 2023, Oracle and/or its affiliates / Public

Chapter 2 details the use of Git as a version control system in more detail.

Fast Feedback Loops

One of the issues inherent with long software lead times is the absence of feedback until it is often too late. It is not hard to imagine a case where 2 teams work on their respective features in separate branches for weeks on end. When it comes to integrating these 2 branches into the release branch it is not uncommon to see merge conflicts. Given the time passed, the number of conflicting changes that occurred in each branch can take a long time to resolve, delaying the release unnecessarily.

Rather than spending weeks before difficulties are detected it might be easier to combine changes from different branches more frequently.

Fast feedback loops are often referred to when it comes to pipeline execution. If it takes too long for a pipeline to run, developers might get frustrated with the process and might start circumventing the use of the pipeline. This is a big "No" in CI as it has a severe impact on the code quality. DevOps engineers must therefore ensure that the pipeline execution runtimes remain short. This can be challenging especially when the provisioning of a test database is part of the pipeline execution.

Chapter 4 discusses various options on how to provide an Oracle Database environment to the CI pipeline so that linting, code coverage, and unit/system/integration tests can be completed in the shortest time possible for your project's CI pipeline execution.

Automated Testing

Integrating code frequently is one step towards a higher release frequency. However, you cannot have confidence in your application if all you did was to test if the code merges (and compiles) without errors. Only testing the deployed application can ensure that new functionality works as expected. Developers embracing **Test-Driven Development (TDD)** know that each new piece of functionality in the application code must be accompanied by a series of tests.

Small and Frequent Changes

The risk of introducing bugs tends to be proportional to the size of the change. A major change that hasn't been merged into the main release branch for weeks carries a larger risk of creating merge conflicts than a small change that's been merged within e.g. a few hours.

Merge conflicts in this context have a direct influence on the team's ability to ship a feature: until the merge conflict is resolved the feature cannot go live. The idea with CI/CD however is to have the software in a ready state at any time so that it could be deployed at the drop of a hat.

New development techniques such as **Trunk-Based Development** might help teams to deliver small, incremental changes more safely. Trunk-based development recommends committing frequently, perhaps even multiple times per day, therefore avoiding the creation of long-lived branches and their related problems.

Automated Deployment

In a database project there is no single build artifact as in a Java project, for example. The application isn't built into a JAR file, WAR file, or container image. With database projects, every "build" might include a deployment to a database.

Typically, you find there is a need to deploy at least twice in database projects:

1. Deployment to a (clone of the) dev environment.
2. Deployment to a higher tier, including production.

For a project to be successful with CI/CD, the deployment mechanism must be identical no matter what environment you are deploying against. The deployment must also be repeatable in a safe manner. In other words, the environments must be so similar – or ideally identical – that a new deployment does not result in errors that were not detected during the CI phase. In environments where a lot of administrative work is manually

Copyright © 2023, Oracle and/or its affiliates / Public

performed, there is a high risk that a deployment to a development environment has almost no resemblance to UAT, Integration, or production environments. The cloud offers help by means of Infrastructure as Code (IaC): using tools like Terraform and Ansible it is a lot easier to create and maintain identical environments. Cloud backups can be restored quickly, reducing the amount of time needed to provision database clones.

Chapter 3 discusses how to create repeatable, idempotent deployments of database schema migrations.

Summary

Many database-centric application projects are handled differently compared to stateless software projects merely generating artifacts, like for example JAR files in Java or executables for other languages. This paper aims to change this situation by introducing the benefits of automation, source code control, and modern development techniques. The following chapters dive deeper into the various aspects, from using version control systems to deploying schema changes using migration tools in CI/CD pipelines.

Copyright © 2023, Oracle and/or its affiliates / Public

Using Version Control Systems

This section covers **version control systems** (**VCS**), particularly Git, the most common version control system at the time of writing. The references to Git should not be interpreted as a general endorsement of that particular technology. The principles described in this paper can be applied to and with any version control system.

No CI/CD without the use of a version control system!

You read in the previous chapter that a **CI** (**Continuous Integration**) server is a central part of the automation architecture, coordinating the execution of scripts, tests, and all other operations via pipelines. Almost all CI servers expect source code to be provided in a Git repository.

You cannot develop a CI/CD pipeline without storing your code in a version control system!

Subjecting your project's source code to version control is the first step towards automating your development and potentially deployment processes. In this chapter, you can read about the various options available to you.

Getting the team's buy-in to using a version control system

Traditionally, (frontend) developers and database administrators (DBAs) have been part of separate teams. While the separation has worked well for a long time, the approach is less suitable for more modern development models. In fact, it hasn't been for a while.

Rather than developers throwing their application code over the proverbial fence, better methods can and should be used. The **DevOps** movement embraces cooperation between developers (the "dev" in DevOps) and operations ("ops"). Although DevOps is more of a change in culture than technology, introducing a new style of cooperation typically entails automating processes that previously were performed manually. That is where **CI** comes back into play.

Introducing a DevOps culture can be difficult, and management must get sufficient buy-in from everyone involved. Otherwise, the modernisation project could encounter more severe difficulties than necessary early on.

Introducing a version control system requires everyone working on the project to share their contributions in a VCS repository. That can imply greater visibility of an individual's contributions, something not everyone is comfortable with. Project members reluctant to work with VCS can be convinced by pointing out the benefits of VCS, including (but not limited to):

- Recording a file's history from the time it was added to the repository.
- The ability to revert to a previous, well-known state.
- Visibility of each code change.
- Enabling distributed workflows.
- Powerful conflict resolution.
- Protection from data loss when using a central repository.
- Much better developer experience compared to storing files locally on a computer or network file share.

For any team pursuing the use of **CI/CD pipelines,** there is practically no alternative to using VCS. This message works best if conveyed gently, taking the concerns of team members on board and offering support, training and mentoring to those unfamiliar with VCS. You could even go so far as to appoint a "VCS Champion" to act as a central point of contact until everyone is comfortable with the new way of working.

Introduction to Git

Git is a *distributed* version control system with a proven track record. Its primary purpose is to allow developers to work on a software project concurrently. It supports many different workflows and techniques and - crucially - helps the team track changes over time. It is also very efficient, storing only those files that have changed in a commit. Git works best with text files; binary files are less suitable for storing in Git. It is also an industry best-

Copyright © 2023, Oracle and/or its affiliates / Public

known method not to store anything created as part of the build, such as Java JAR files or compilation results (commonly referred to as build artifacts) in Git.

Git has many features contributing to a great developer experience. In addition to the command line interface, all major Integrated Development Environments (IDE) support Git out of the box.

Git Terminology

Your project's files are part of a (software) *repository*. In addition to your project's files, the repository contains the meta-information required for the proper working of Git. That includes internal data and configuration information, such as a list of files to ignore, local branch information, and so forth.

*It is imperative **NOT** to commit sensitive information such as passwords or secure tokens in Git!*

Git can operate on *local* and *remote* repositories. The local repository is typically *cloned* from a centrally hosted Git Server such as GitLab, GitHub, or any other system in your organisation. Developers use the local copy to make changes before uploading ("pushing") them to the central ("remote") repository.

If a remote repository does not yet exist, an administrator needs to create the remote repository first and grant the necessary privileges to the team members to clone from and contribute to it.

All files in a repository are associated with a *branch*. Branch names are arbitrary; most projects follow their own nomenclature. Conventionally, *MAIN* is considered to be the stable branch, although no rule enforces that name. Branching is one of the core features in Git, but excessive branching has been found to cause problems; see below for a more detailed discussion.

Files newly added to the repository start out as *untracked* files. Existing files that haven't been edited yet are said to be *unmodified*. Changes to files in the directory aren't automatically saved to the repository. New files must be *added* to the repository first whilst changed files must be *staged* by adding them to the staging area. Once files are added or staged, they can be committed to the project.

After the commit completes, all the files that have been committed to the repository have their status set to unmodified.

Each commit requires you to add a commit message. The message is an essential commit attribute: ideally, it conveys the nature of the change as precisely as possible. Here is an example of a useful commit message:

```
(issue #51): extend column length of t1.c1 to 50 characters
```

Good commit messages help to understand the commit history tremendously.

Branching and Merging

Branching and merging, both resource- and time-intensive processes with previous generations of VCSs, are no longer an area of concern with Git.

Recent research, however, points out that extensive branching is likely to lead to hard-to-resolve and time-consuming merge conflicts, potentially introducing a significant delay, which prevents you from releasing changes at a faster rate. Small, incremental changes allow organisations to release far more frequently. Taken to the extreme, some models propose using a single branch or trunk against which developers submit their code. This approach is known as Trunk-Based Development.

Once a feature is ready, developers typically create a **Pull Request** (**PR**)/**Merge Request** (**MR**) to merge their feature into the MAIN branch.

Pull and Merge Requests

Originally, Pull Requests (GitHub), and Merge Requests (GitLab) weren't part of Git. They have been introduced as part of the hosted development platform and have enjoyed widespread adoption.

The goal of these is to notify the maintainer of a given branch of the submission of a new feature or hotfix. Metadata associated with the MR/PR typically involves a description of the problem, links to a collaboration tool,

Copyright © 2023, Oracle and/or its affiliates / Public

and various other workflow details. Most importantly, all the commits from the source branch and all the files changed in it are listed, allowing everyone to assess the impact and comment on the changes.

Code Reviews are often performed based on Merge Requests, especially in Open Source projects (see below).

Forking a Project

Forking is less common in in-house software development projects than with Open Source Software (OSS). OSS encourages contributions to the code, but for obvious reasons, these contributions need to undergo a lot of scrutiny before they can be merged.

In other words, regular users don't have write privileges on public software projects hosted on GitHub or GitLab. To get around this limitation, developers wishing to contribute to the project create a copy, or a *fork*, in their own namespace and modify it as if it were theirs. To integrate the changes back into the original project, the next step is to raise a Pull Request (GitHub), or Merge Request (GitLab) once the contribution is ready.

Project maintainers can then review the contributions and either merge them or request further changes or enhancements. Once the contributions have been merged, the contributor's fork becomes redundant and can be archived, deleted or synced with the original project repository for future contributions.

Your Database Project's Git Repository

As with every aspect of the software development lifecycle, spending some time thinking about the future before starting the implementation pays off. Mistakes made early in the project's lifespan can prove costly and complex to resolve later.

Single Repository vs Separate Frontend and Backend Repositories

There is an ongoing discussion in the developer community about whether application code like that of your Angular, React, or any other frontend technology should coexist with database change code inside a single repository. For most modern applications, especially those following a micro-service pattern it makes a lot of sense to include frontend and backend code in the same repository.

For existing, complex software projects, especially those where the database is accessed by a multitude of applications, creating a separate Git repository might make sense. There is a risk of introducing delay in the release cadence if the application and database repositories are separate: changes required by the application might not be incorporated in time, causing delay. The development workflow must ensure that there aren't siloed team structures in place, or carried over.

This tech brief was written under the assumption that developers own both the frontend and backend, therefore combining both the user interface and the database schema changes in a single repository.

Directory Structure

When it comes to your database project's directory structure, the file layout choice matters a lot. It primarily depends on the method used for deploying changes:

- Migration-based approach ("delta" or "increment" method)
- State-based approach ("snapshot" method)

Using a **migration-based approach**, developers deploy their changes based on the expected state of the database schema. Assuming a change at `t0` deployed a table in the schema, the following database change uses the `ALTER TABLE` statement to modify the table. The schema migration is a continuous process where one change depends on the previous one.

Using a **state-based approach**, developers declare the target state, like for example the structure of a table. The deployment tool assesses the current state of the table and creates a set of changes on the fly transitioning the current state to the target state.

This tech brief suggests a method where both are combined. However, the state is stored only for reference or to be used to populate an empty schema.

Copyright © 2023, Oracle and/or its affiliates / Public

Remember that **with stateful applications every release is a migration**. As part of a release, existing schema objects like tables are modified. You may need to add a new column, or maybe change the column definition for example. These operations are performed using the `ALTER` … SQL statement.

You will read in Chapter 3 that many schema migration tools deploy code based on the assumption that scripts are immutable. This is different compared to many other software projects where source code is edited to accommodate new requirements. Using schema-migration tools you typically create a new, additional script to change the existing state of schema objects; changes are carried forward, and existing scripts are never changed.

The following directory structure allows for a combination of the migration-based and state-based approaches. It has been created with a single repository in mind for frontend and database changes. Since there are too many different directory structures possible, depending on the frontend language or framework (Spring, Flask, React, Angular, Vue, etc.) the tech brief focuses on the database part exclusively. It is stored in the `src/database` directory, found in the project's root.

```
$ tree src/database/
src/database/
├── controller.xml
├── r1
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r2
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r3
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
└── utils
    └── newRelease.sh
```

One directory exists per *database release*, named `r1` through `r3` in the example. Remember from the previous discussion that database change scripts are immutable, new requirements are implemented using a new release folder. The name of the release folders above is just an example. Any naming convention can be chosen as long as it clearly and uniquely identifies a database release.

Database changes are stored as files inside the `migrations` folder under the given database release folder, and in the case of Liquibase deployments, these files are executed in alphabetical order if no explicit order has been defined.

The `state` directory contains the schema DDL for all objects to produce a schema structure exactly in the way it should look like at the end of the successful schema migration. This directory can be used as a reference and for troubleshooting. It should **never** be used for schema change deployments.

Unit tests sometimes require the use of test data. This test data set can be stored in the `testdata` directory, if needed.

The optional `utils` directory stores utility functionality that doesn't need to be applied to the database as part of the release. It typically contains scripts to perform pre- and/or post-deployment validation but **never** is such a script allowed to perform changes to the database schema.

Copyright © 2023, Oracle and/or its affiliates / Public

As soon as a database release has been applied to a database, no further changes are made to the files in its `migration` folder. Think of new releases as resuming the process from the previous release and applying a set of changes to the schema.

An optional `utils` directory under `src/database` can be used to store other common utility functionality that may be used to simplify the process of creating and validating new database releases, etc. but **never** to perform schema change deployments.

A helper script named `newRelease.sh` creates the necessary folder structure and other scaffolding work, like adding the release to the main application changelog (more on that topic later) used by Liquibase.

Release Management

As you read in the previous section, each database release is maintained in its own, dedicated sub-directory in the repository to avoid conflicts. Migrating the application from one release to another merely requires the execution of the migration scripts. You are not limited to making a jump from release `rN-1` to `rN`; you can also start earlier, for example, at `rN-5`. Thanks to the aforementioned schema migration tools, only those scripts that haven't been applied to the database yet will be applied as part of the database release (= migration).

Determining whether an application (schema) is in the "correct" state might be difficult using this approach. If release management wants to check if the migration was successful, they need to know exactly what the schema should look like. That can be difficult using the migration-based approach because you have to identify which changes were made to a schema object across at least one release, or more.

Storing a reference to compare against simplifies that task. That is where the `state` directory might be helpful: it should contain the DDL statements for each schema object at its desired state. Using these DDL statements, it is trivial to compare the schema object in the database against the state it should be in.

Integrating with the Continuous Integration Pipeline

It is common practice to trigger your project's CI pipeline's execution after a commit has been pushed to the remote repository. Hence it is very important to only push code that is expected to work with a reasonable level of confidence. Local testing and commit hooks can help build that confidence. Should the pipeline's execution abort due to whatever circumstances, an all-hands situation arises, and everyone should work on fixing the pipeline at a high priority.

You can read more about CI pipelines in a later section of this tech brief.

Summary

Git is the most common version control system in use. Understanding how Git works and getting everyone's buy-in to using Git are crucial first steps into adopting a modern software development architecture. Whether you use separate repositories for database code and frontend code or not depends entirely on your project and its surrounding circumstances.

Choosing the "correct" directory layout for a database project up-front pays dividends once the project is well underway. Schema changes should be applied using dedicated, commercial-of-the-shelf (COTS) tools like Liquibase to avoid the cost of maintaining a home-grown solution.

Copyright © 2023, Oracle and/or its affiliates / Public

Ensuring Repeatable, Idempotent Schema Migrations

The previous chapters have put great emphasis on the need for using version control systems as part of the development process. They deliberately did not cover how and what format to use when creating schema migration scripts in detail. This will be addressed in this section.

Remember from Chapter 2 that every database release is a schema migration. Unlike with other software development projects, existing database code cannot be touched for reasons explained in this chapter. This is perhaps the single biggest difference between stateless application development and database projects. This chapter explains the why, and how to deal with this situation.

Using a directory structure such as the one shown below can prove beneficial for database releases. It is based on the fact that a single Git repository is used for both the application's frontend and backend code. The database part of the project can be found in `src/database` under the root directory of the project. This is merely a suggestion and can be deviated from as long as the database part is self-contained:

```
$ tree src/database/
src/database/
├── controller.xml
├── r1
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r2
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
├── r3
│   ├── migrations
│   ├── state
│   ├── testdata
│   └── utils
└── utils
    └── newRelease.sh
```

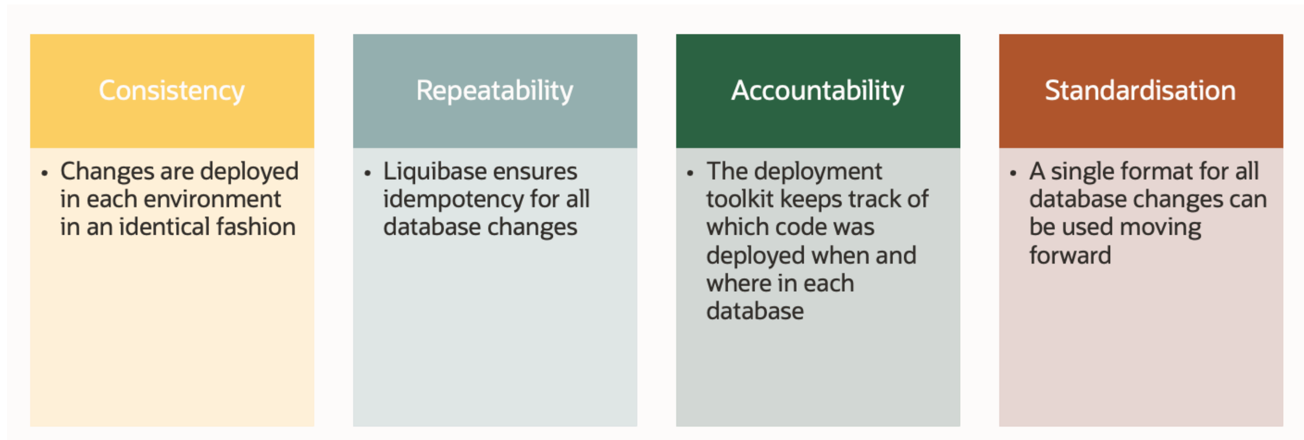
The initial software release is found in the `r1` directory. Once the first release has been deployed, any further changes to the database schema should be provided in a new release folder, like `r2` in the above example.

Benefits of using Dedicated Schema Migration Tools

Many teams use schema migration tools such as Liquibase, Flyway, or comparable ways of deploying schema migrations built in-house. Liquibase, for example, offers the following advantages:

Copyright © 2023, Oracle and/or its affiliates / Public

Figure 2. Advantages of using Liquibase to deploy schema changes



Using off-the-shelf tools is generally preferable to using home-grown software. The maintenance effort required to keep the custom solution up-to-date, across all environments, does not typically add value and might draw precious development resources away from delivering the actual product.

Using SQLcl and Liquibase to deploy Schema Migrations

Oracle **SQL Developer Command Line** (**SQLcl**) is a free command line interface for Oracle Database. It allows you to interactively or batch-execute SQL, PL/SQL, and JavaScript. SQLcl provides in-line editing, statement completion, and command recall for a feature-rich experience, all while also supporting your previously written SQL*Plus scripts.

Liquibase is an Open Source database-independent library for tracking, managing, and applying database schema changes.

Combined, these two technologies provide a great way for deploying database migration scripts inside CI/CD pipelines. Using checksums and other metadata, Liquibase and other comparable tools, such as Flyway, can identify which script has been run against a database, avoiding an unnecessary and potentially harmful redeployment.

Perhaps the greatest advantage of SQLcl is its low storage footprint, its plentiful modern command line features and the built-in Liquibase functionality. This tight integration is a great productivity boost, especially if you are targeting systems with mutual TLS encryption enabled, such as Oracle Autonomous Database.

Liquibase integration in SQLcl is documented in the main SQLcl documentation. It is unique and different to that of the Liquibase Open Source edition found on the web, by having been enhanced by Oracle to make Liquibase aware of Oracle Database advanced functionality such as extended data types, edition-based redefinition (EBR) and more. It is highly recommended to use the built-in Liquibase support inside SQLcl when working with Oracle Database, instead of the Liquibase Open Source edition.

Liquibase Terminology and Basic Concepts

Before you can start working with Liquibase you need to understand the basic concepts first. These include:

- Changelog
- Changeset
- Change Type

The following figure provides more detail concerning each of these. Please refer to the official Liquibase documentation for all the details.

Copyright © 2023, Oracle and/or its affiliates / Public

Figure 3. Liquibase changelog, changeset and change type explained

Changelog	Changeset	Change Type

• A collection of change sets to be made to the database.
• "The Release"
• SQLcl can generate a changelog for:
  • Entire schemas
  • Database Objects
  • ORDS REST APIs
  • APEX applications
	• Liquibase organizes a change to the database in changesets.
	• A changeset contains a single change to the database, depending on the change type.
	• Changesets can be provided in XML, JSON, YAML, and SQL formats
		• Each changeset must specify the nature of the work to be performed.
		• The change type defines which operation the changeset performs, such as creating a table, adding a column, etc.

Changesets are the basic entities developers create as part of a new database release. Multiple changesets are referred to as your database's changelog. A changeset contains one or more change types. As a best practice, limit yourself to one type of change per changeset to avoid complications during deployments.

As per the above figure, changesets can be provided in many formats. SQLcl will default to the XML format when reverse-engineering schema objects. This is great for smaller projects or when you are just starting out with SQLcl and Liquibase. The downside to using the XML format is the inability of most linters to read the actual command embedded in CDATA tags. If linting your code is important to your project, you should consider using the SQL format instead. Using the SQL format might also be the easiest way to transition into the use of Liquibase as it merely means decorating existing SQL scripts with Liquibase-specific annotations.

Practical Aspects of Creating the Database Changelog

Developers typically create multiple files containing their changesets, like creating a table, adding columns to a table, or deploying database code in JavaScript or PL/SQL. These changesets are then included in the changelog to simplify deployment. The changelog is also known as "the release".

What does this look like in a project? Let's assume a new column must be added to an existing table, and the column to be added is a foreign key. The changelog is comprised of the following three changesets:

1. Addition of a column to the table.
2. Creation of an index covering the new column.
3. Addition of the foreign key on the new column to the parent table.

A developer typically creates a separate file for each of the above-mentioned changesets. To simplify deployment, an additional file is frequently created. In contrast to the first 3 files the latter, often called the main changelog, or release changelog, doesn't contain any database-specific commands to create schema objects or interact with them. It rather lists all the files to be deployed by Liquibase using the `include` or `includeAll` directives. You can read more about all of these later in this chapter.

Formatting your changelog

To be used with Liquibase, changesets must be annotated. At the very least they need to have an author and ID associated. The combination of ID, author as well as the file path of the file itself uniquely identifies a changeset. This is important when it comes to determining if a changeset has already been executed against a database.

In Oracle Database, PL/SQL code is typically terminated by a forward slash at the end of the program, something Liquibase needs to be told about. Adding `endDelimiter:/` as an additional attribute makes the tool aware. Otherwise, any semi-colon in the PL/SQL code would act as a statement terminator and cause an error. Furthermore, Liquibase removes any comments from changesets by default. In most cases, this is not desirable. Providing the `stripComments:false` attribute solves this problem.

The following example demonstrates how to use the Liquibase attributes for a changeset provided as SQL:

```
--liquibase formatted sql
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
--changeset developer1:"r1-01" failOnError:true labels:r1

CREATE TABLE sessions (
    -- this is a UUID
    session_id          char(32) not null,
        constraint pk_session_hit_counter
            primary key (session_id),
    browser             varchar2(100),
    operating_system    varchar2(100),
    duration            interval day to second
);
```

As you can see in the example, the SQL file must feature the Liquibase preamble (`--liquibase formatted sql`), which tells Liquibase that the file format is SQL (instead of XML or JSON) followed by the changeset information itself (`--changeset author:ID`, e.g. `developer1:"r1-01"`). Additional flags for the changeset can be provided, out of which `failOnError` and `labels` are the most important ones. Setting `failOnError` to TRUE (together with SQLcl's `whenever sqlerror exit` command) ensures that the pipeline's execution halts as soon as an error is encountered. Adding a label corresponding to the release name allows you to provide label filters when running code, executing changesets with a given label exclusively.

See below for further explanations on how to ensure the pipeline's execution halts in case of errors.

PL/SQL changesets can be written along the same lines, albeit additional attributes for the changeset are required or desirable. In this example, you see a unit test based on utPLSQL. The processing logic in utPLSQL is heavily based on comments. (Note: the "↲" character in the examples just indicates that in the file itself the text continues on the same line but is wrapped into another line in the example below due to the limited page width).

```
--liquibase formatted sql
--changeset developer2:"r1-09" failOnError:true endDelimiter:/ stripComments:false ↲
labels:r1

create or replace package my_test_pkg as

    --%suite(unit tests for my application)

    --%test(verify that something is done right)
    --%tags(basic)
    procedure my_test_001;

end my_test_pkg;
/
```

Here you can see the same `author:ID` combination and `label` attributes like in the previous example. Additionally, `stripComments` attribute is set to false to ensure none of the comments inside the PL/SQL code are removed, which would otherwise break the utPLSQL unit test., and that the SQL command is defined to terminate with the `/` character, as instructed by the `endDelimiter` attribute, as Liquibase would otherwise terminate the SQL command at the first occurrence of a `;` character inside the PL/SQL package.

Deploying the Changelog

Code changes are deployed using the `lb update` command in SQLcl. Liquibase then reads all the changesets in the provided changelog in order.

Before applying a given changeset, a metadata query against the changelog table is performed. Should the script have run previously, it is skipped (unless the `runAlways` attribute is set, which is **not** recommended).

Copyright © 2023, Oracle and/or its affiliates / Public

Liquibase's deployment logic can be used to great effect. A "main" changelog can be defined once for the project. Rather than keeping it up to date with all the file changes, it is possible to use the `includeAll` directive as shown in this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
      "http://www.liquibase.org/xml/ns/dbchangelog
       http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-latest.xsd">
  <includeAll path="r1/migrations"/>
  <includeAll path="r2/migrations"/>
</databaseChangeLog>
```

Each new database release (`r`*`N`*) is added using a new `includeAll` directive. All the Liquibase changesets found in the directory indicated are executed in alphabetical order. Make sure you place the files in the directory prefixed with a number, for example, to ensure they are executed in the correct order.

Circling back to the previous example (adding a foreign key column to a table), the developer creates the following files in `r`*`N`*`/migrations`:

- 01_*tablename*_add_column_*columnname*.sql
- 02_*tablename*_add_index_*indexname*.sql
- 03_*tablename*_add_foreign_key_*fkname*.sql

This way the order of execution is guaranteed to be identical and in the right order. To ensure that all changes in `r`*`N`*`/migrations/*` are executed an entry referring to the directory as added to the main changelog.

Thanks to the metadata checks performed by Liquibase, any changeset found in the baseline and migrations folder that has already been executed is skipped automatically; no conditional logic is required. Since the main changelog file itself doesn't include any changesets and hence no SQL commands on its own, it is fine to use XML or another supported format.

In addition to the metadata query performed initially, another safety net exists preventing broken releases. An MD5 checksum is maintained for each file to ensure it hasn't been tampered with. The checksum is stored as part of Liquibase's metadata catalogue. The next time `lb update` is attempted, the change is noticed and Liquibase aborts its run.

To prevent issues caused by changed sources during deployment it is possible to validate the changelog prior to executing it. The `lb validate` command verifies the changelog and warns if checksums changed.

Not being able to fix code in place due to the requirement to keep files immutable raises an interesting question: how can a problem like a bug in a code unit be fixed? Following the Liquibase philosophy such a case requires *forward fixing.* In other words, creating a new release with the fixes necessary to correct the bug. Removing the bug in the source code file and re-deploying it is not a recommended option.

Furthermore it is possible to rollback changes in Liquibase, a feature not in scope of this tech brief.

Checking the status of your deployments

The metadata used during Liquibase deployments is available to everyone. The target schema features several tables that are created by Liquibase:

- DATABASECHANGELOGLOCK
- DATABASECHANGELOG_ACTIONS
- DATABASECHANGELOG

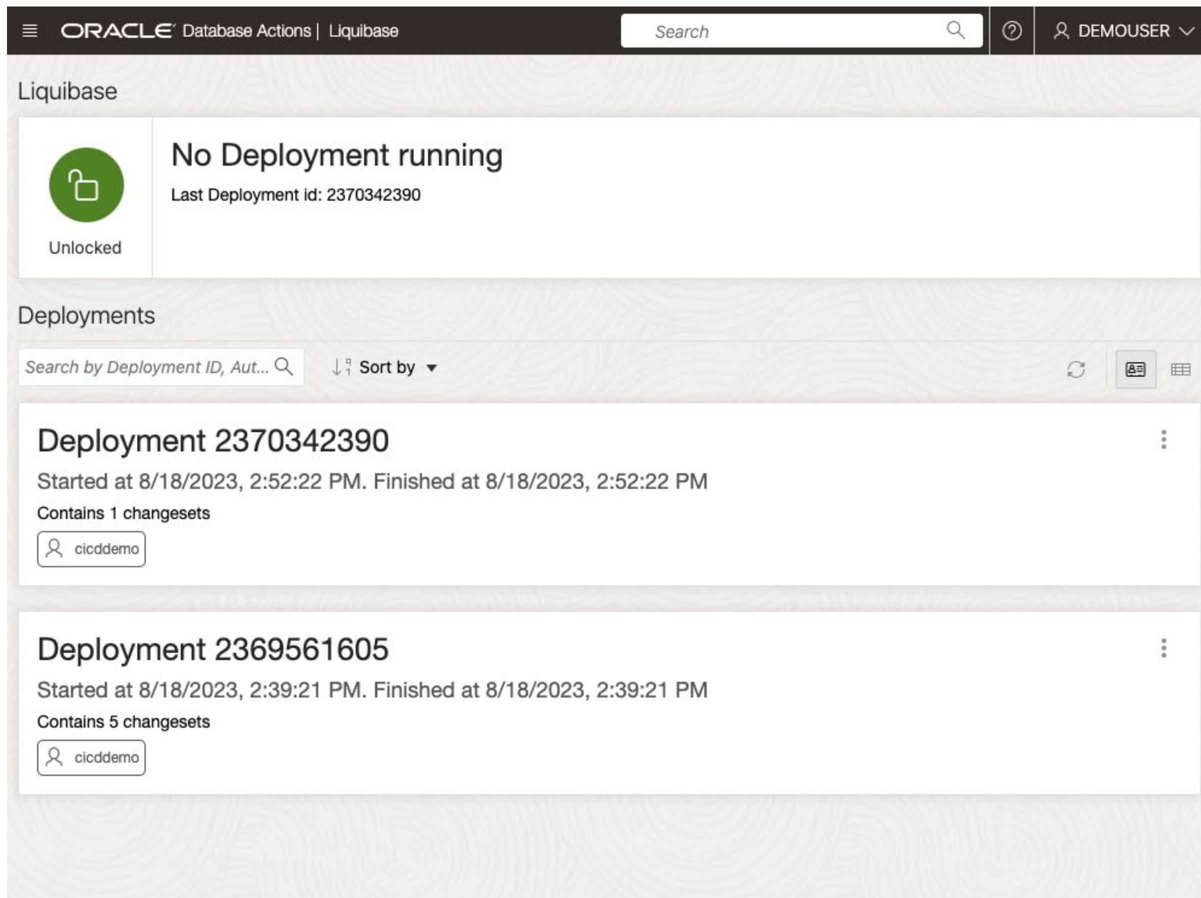
Copyright © 2023, Oracle and/or its affiliates / Public

The `lb history` command exposes the contents of these tables to release management. Here is an example from an actual deployment against a database:

```
SQL> lb history
--Starting Liquibase at 14:39:27 ↲
(version 4.17.0 #0 built at 2022-11-02 21:48+0000)

Liquibase History for
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=cidb.test.oraclevcn.com)(PORT=152
1))(CONNECT_DATA=(SERVICE_NAME=tcdae8c8)))
- Database updated at 8/18/23, 2:39 PM. Applied 5 changeset(s) in 0.058s, ↲
DeploymentId: 2369561605
  r1/migrations/01_javascript_sources.sql::migration00001-01::cicddemo
  r1/migrations/02_sessions.sql::migration00001-02::cicddemo
  r1/migrations/03_hit_counts.sql::migration00001-03::cicddemo
  r1/migrations/05_hit_counter.sql::migration00001-05::cicddemo
  r2/migrations/02_hit_counter.sql::release00002-01::cicddemo

- Database updated at 8/18/23, 2:52 PM. Applied 1 changeset(s), DeploymentId: 2370342390
  r2/migrations/03_hit_counter_pkg.sql::release00002-02::cicddemo


Operation completed successfully.
```

Database Actions provides a graphical frontend to the text-based version as demonstrated in the following screenshot:

Figure 4. Screenshot showing Database Action's Liquibase deployment screen



Copyright © 2023, Oracle and/or its affiliates / Public

Whichever way you choose to track your deployments – using the command line or a graphical user interface - you will always know which changeset has been deployed against the database, and when. This is a great step forward for most users as it requires a lot less coordination between the teams. Tracking database changes is now a by-product instead of an important goal during application development, and no effort is required at all to enable this type of tracking. This built-in benefit is not to be underestimated.

Ensuring Pipeline Failure

As you read in an earlier chapter, it is imperative for the CI pipeline to fail in case database deployment errors are encountered. In the case of Liquibase you need to set a few attributes to achieve this goal.

First, you should always set `failOnError` to TRUE. This will ensure SQLcl aborts the execution of the current changeset. The pipeline's logs can then be used to find out why the problem occurred in the first place.

Second, SQLcl must also be instructed to fail on error for the pipeline execution to abort. Just like SQL*Plus, users can specify the `whenever sqlerror exit` command.

Many users chose to create their own deployment script in the `src/database/utils` subdirectory of their project. For example, such a deployment scripts could create a restore point inside the database prior to the deployment, set a Liquibase tag in case rollback operations are desired, and then deploy the changelog. Because SQLcl has built-in Liquibase support, all of this can be executed in a regular *.sql file like it were any other SQL command. An example deployment script is shown here:

```
/*
    NAME:
        deploy.sql
    PURPOSE:
        enable liquibase deployments in CI/CD pipelines

    PARAMETERS
        (1) tag name (used for lb tag and creating a restore point)
            typically the COMMIT SHA. Since that can start with an invalid
            character the value is prefixed with a t_.
            This parameter is provided during the invocation of deploy.sql
            as part of the pipieline execution.
*/
whenever sqlerror exit

declare
    nonexistant_restore_point exception;
    pragma exception_init(nonexistant_restore_point, -38780);
begin
    if length('&1') = 0 then
        raise_application_error(
            -20001,
            'Please provide a valid tag name!'
        );
    end if;

    -- drop the restore point, it doesn't matter if it exists or not.
    -- note it's not possible to use bind variables in dynamic SQL
    -- executing DDL statements
    begin
        execute immediate
            replace(
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
                'drop restore point :MYRPNAME',
                ':MYRPNAME',
                dbms_assert.simple_sql_name('t_&1')
            );
    exception
        when nonexistant_restore_point then null;
        when others then raise;
    end;
end;
/

create restore point t_&1;
lb tag -tag t_&1
lb update -changelog-file controller.xml
drop restore point t_&1;
```

This file can be used in the CI pipeline. As soon as an error is encountered as part of the schema migration, the pipeline will stop. Remember to monitor the use of restore points and clean them up once the release has been signed off or otherwise marked as successful.

Summary

Using schema migration tools such as SQLcl and Liquibase or Flyway allows developers to be more confident about their database migrations. Once they embraced the workflow associated with each tool, schema migrations become a lot more manageable. Combined with the mantras of releasing often, and making small, incremental changes, there does not have to be a situation where the main changelog comprises hundreds of changes. This would indeed be an uncomfortable situation as users might face a pipeline timeout. Therefore, it is also important to test against production-like volumes of data, something that will be covered in the next chapter.

Copyright © 2023, Oracle and/or its affiliates / Public

Efficient and Quick Provisioning of Test Databases

Test databases play an essential part in **Continuous Integration** (**CI**) pipelines. In this context, databases are often referred to as **CI databases**. As you read in Chapter 1 of this tech brief several tests are automatically run once the release has been deployed into the CI database. Ideally, the entity – for example a database schema, a Pluggable Database (PDB), or a cloud service – represents the production database.

Following the general rule that a CI pipeline's execution must quickly finish, the time it takes to complete the provisioning of the deployment target must be as short as possible. Remember that fast feedback is essential for the efficient use of CI/CD pipelines. The sooner a developer knows about an issue, the sooner it can be fixed.

There are different approaches available to shorten the creation of a CI database:

- Provisioning an Autonomous Database Cloud Service.
- Use of container images (stand-alone/orchestrated by Kubernetes).
- Creation of a Pluggable Database.
- Using Copy-On-Write technology to clone a (pluggable) database.
- Provisioning a database schema.

Each of these techniques offers advantages and disadvantages, to be discussed in this section.

Autonomous Database

Oracle Autonomous Database provides an easy-to-use, fully autonomous database that scales elastically and delivers fast query performance. As a cloud service, Autonomous Database does not require database administration.

Autonomous Database-Serverless (ADB-S) databases are a good candidate for customers with an existing cloud footprint. They are great for use in CI pipelines because of their high degree of automation and the many different options available to create them. Common options to create Autonomous Databases include:

- Creating an empty ADB-S instance (less common).
- Cloning an ADB-S instance, for example, from production.
- Creating an ADB-S instance from a backup.

All these operations can be automated using **Terraform**, the **Oracle Cloud Infrastructure** (**OCI**) Command Line Interface (CLI), or even plain **REST** calls. The following Terraform snippet provides a minimum of information required to clone an existing Autonomous Database for use in the CI/CD pipeline:

```
resource "oci_database_autonomous_database" "clone_adb_instance" {
  compartment_id              = var.compartment_ocid
  db_name                     = var.ci_database_name
  clone_type                  = "FULL"
  source                      = "DATABASE"
  source_id                   = ci_database_autonomous_database.src_instance.id
  admin_password              = base64decode(local.admin_pwd_ocid)
  cpu_core_count              = 1
  ocpu_count                  = 1
  data_storage_size_in_tbs    = 1
  nsg_ids                     = [ module.network.cicd_nsg_ocid ]
  subnet_id                   = module.network.backend_subnet_ocid
}
```

The ADB-S instance is created within a private subnet (created by a Terraform module, not shown here), and integrated with the CI Server, the pipeline's infrastructure and Network Security Group (NSG). This snippet creates

Copyright © 2023, Oracle and/or its affiliates / Public

a full clone, there are additional clone types available. You should pick the one that best matches your needs for your workload.

You can read more about cloning Autonomous Database in the official documentation set.

Using Container Images

For the past decade, container technology has become ubiquitous. Unlike a classic virtual machine, they offer less isolation from one another but at the same time they are much more lightweight.

The appeal of using container technology is simplification: CI/CD pipelines very often build container images. These container images can be deployed anywhere where a container runtime is available, from a developer's laptop all the way up the tiers into production. The container images are self-contained and thanks to the process of packaging runtime libraries together with the application code, you are less likely to run into deployment issues.

For many customers, using container images has become the norm: if the application is deployed in a container, why not use a database container as well? Oracle's own container registry features a section dedicated to Oracle Database.

Copyright © 2023, Oracle and/or its affiliates / Public

Figure 5. Screenshot showing Database container images on Oracle's container registry

Oracle Container Registry: Exp

container-registry.oracle.com/ords/f?p=113:1:128898806357580:::1:P1_BUSINESS_...

ORACLE Oracle Container Registry Sign In

Database Repositories

enterprise	Oracle Database Enterprise Edition
gsm	Oracle Global Service Manager
instantclient	Oracle Instant Client
rac	Oracle Real Application Clusters

1 - 4

Repository	Description	Open Source License Text
express	Oracle Database Express Edition	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
free	Oracle Database Free	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
observability-exporter	Oracle Database Observability Exporter (Metrics, Logs, and Tracing)	The container image you have selected and all of the software that it contains is licensed under UPL that are provided in the container image. Your use of the container is subject to the terms of those licenses.
operator	This image is part of and for use with the Oracle Database Operator for Kubernetes	The container image you have selected and all of the software that it contains is licensed under UPL that are provided in the container image. Your use of the container is subject to the terms of those licenses.
ords	Oracle REST Data Services (ORDS) with Application Express	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
otmm	Oracle Transaction Manager for Microservice	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
sqlcl	Oracle SQLDeveloper Command Line (SQLcl)	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.

1 - 7

ORACLE Integrated Cloud Applications & Platform Services

About Oracle Legal Notices Terms of Use Your Privacy Rights

Copyright (c) 2017, 2018, Oracle Corporation. All Rights Reserved.

Please refer to My Oracle Support *Oracle Support for Database Running on Docker (Doc ID 2216342.1)* for more details concerning database support for the various container runtimes.

Using container images with Podman or Docker

The following example demonstrates how to provision an Oracle Database 23c Free database using the official container image. Note that the database is ephemeral in this scenario, in all other cases you must ensure that you provide a volume to the container or else you might incur data loss. The example was tested on Oracle Linux 8, using the distribution's default container runtime, Podman.

```
podman run --rm -it \
--secret=oracle_pwd \
--name cicd-example \
--publish 1521:1521 \
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
container-registry.oracle.com/database/free:latest
```

The above command starts a new container instance based on the Oracle Database 23c Free image and opens listener port 1521. It initializes both the `SYSTEM` and `SYS` database user passwords to the value stored in a Podman secret named `oracle_pwd`. After less than 1 minute the database is ready to be used and can be accessed on port 1521 on the container host.

Note that your CI/CD pipeline can use the provisioned, empty database as a source, or use cloning technology described later in this chapter to create a copy of an existing "golden copy" (Pluggable) Database. The latter might be the more efficient, in other words: less time-consuming approach.

Using Container Images with Kubernetes

Advanced users of container technology might want to deploy database containers in Kubernetes or a comparable orchestration engine. Instead of manually provisioning and managing Oracle Database containers and Kubernetes cluster resources administrators can make use of the open source Oracle Database Operator for Kubernetes.

As part of Oracle's commitment to making Oracle Database Kubernetes-native (that is, observable and operable by Kubernetes), Oracle released the Oracle Database Operator for Kubernetes (OraOperator). OraOperator extends the Kubernetes API with custom resources and controllers for automating Oracle Database lifecycle management.

The current release (version 1.0.0) supports a multitude of database configurations and infrastructure including support for Autonomous Database . For a complete list of supported operations by database and infrastructure type, refer to the documentation.

OraOperator can be instructed to deploy an instance of Oracle Database 23c Free using the following YAML file:

```
#
# Copyright (c) 2023, Oracle and/or its affiliates.
# Licensed under the Universal Permissive License v 1.0 as shown at
# http://oss.oracle.com/licenses/upl.
#
apiVersion: database.oracle.com/v1alpha1
kind: SingleInstanceDatabase
metadata:
  name: freedb-sample
  namespace: default
spec:
  ## Use only alphanumeric characters for sid, always FREE for Oracle Database Free
  sid: FREE

  ## DB edition
  edition: free

  ## Secret containing SIDB password mapped to secretKey
  adminPassword:
    secretName: freedb-admin-secret
  ## Database image details
  image:
    ## Oracle Database Free is only supported from DB version 23 onwards
    pullFrom: container-registry.oracle.com/database/free:latest
    prebuiltDB: true

  ## Count of Database Pods. Should be 1 for Oracle Database Free or Express Edition.
```


Copyright © 2023, Oracle and/or its affiliates / Public

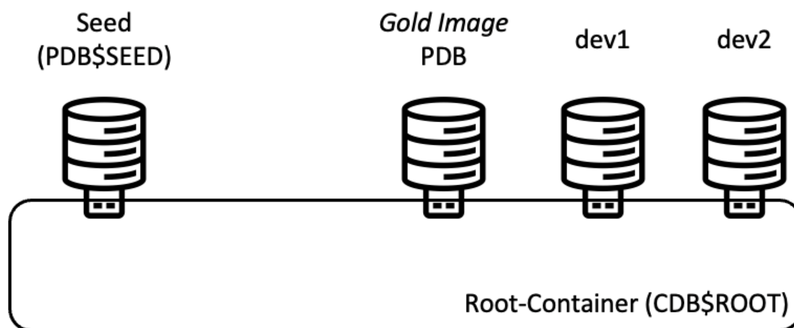
```
  replicas: 1
```

Once you apply the YAML file, OraOperator takes care of the database's setup. You can connect to the database a few moments after submitting the YAML file to the Kubernetes API.

Using Container Databases

Container Databases (CDBs) have been introduced with Oracle 12c Release 1. Unlike the traditional, non-CDB architecture, Container Databases are made up of an Oracle-managed seed database as well as one or multiple user Pluggable Databases (PDBs). A PDB is a user-created set of schemas, objects, and related structures that appear logically to a client application as a separate database. In other words, each PDB provides namespace isolation, which is great for consolidating workloads or providing separate, isolated development environments.

Figure 6. Using Container Databases to quickly provision test environments



Additional levels of abstraction are possible in the form of Application Containers; these are out of scope of this tech brief.

Note that the use of Container Databases might require an extra license, please refer to the Database Licensing Guide of the particular Oracle Database version in question for more information.

The following sections describe popular options for creating Pluggable Databases as part of CI/CD pipelines before discussing ways to automate their creation and deletion.

Creating a new, empty Pluggable Database

A newly created PDB is "empty" after its creation. Your CI pipeline must be able to re-deploy the entire application first before unit tests can be run. This is potentially a time-consuming task. If you are using a container image as described earlier, you may already have a deployment target of an empty PDB. Both, Oracle Database 23c Free and its predecessor, Oracle Database Express Edition (XE) provide an empty PDB out of the box. It is named FREEPDB1 or XEPDB1 respectively. Combined with the setup based on a container image you can spin up and connect to FREEPDB1/ XEPDB1 in less than a minute in most cases.

Cloning an existing Pluggable Database

A more sophisticated, and potentially less time-consuming way of deploying schema changes is the use of Oracle's PDB cloning functionality. Since the inception of Oracle's Multitenant Option in 12c Release 1 more and more ways for cloning PDBs have been added. The topic is covered in detail in Chapter 8 of the Database Administrator's Guide.

Cloning an existing, "golden image" PDB can greatly reduce the time it takes to deploy the application. Provided that appropriate tooling such as Liquibase or Flyway is used, the release can be applied to a PDB clone in as little time as possible. Tooling for managing schema changes is discussed in Chapter 3 of this tech brief.

There are many options for cloning as per the documentation reference above. Using sparse clones (based on Copy-on-Write technology) storage requirements for the cloned PDB are typically reduced by a significant factor.

Copyright © 2023, Oracle and/or its affiliates / Public

Note that the clone of your PDB should resemble production to avoid unpleasant surprises during the production rollout. If you prefer not to run unit tests on a production-sized database clone please consider this during the integration and/or performance testing stages.

Automating Pluggable Database Lifecycle Management

The manual method of requesting a clone of a system via the creation of a change ticket is no longer viable for most users, it simply takes too long. This traditional workflow is also not suitable for use with CI pipelines. A first step towards the use of a CI pipeline consists of automating the PDB lifecycle. Thankfully this task has already been completed: Oracle REST Data Services (ORDS) provides REST endpoints you can use to automate the creation, clone, and deletion of PDBs.

The following example has been taken from an existing CI pipeline. The `curl` utility is used to clone the "gold image PDB":

```
clone-source-PDB:
  stage: build
  environment:
    name: testing
  script: |
    curl --fail -X POST -d '{
      "method": "clone",
      "clonePDBName": "'${CLONE_PDB_NAME}'",
      "no_data": false,
      "snapshotCopy": false,
      "tempSize": "100M",
      "totalSize": "UNLIMITED"
    }' \
    -H "Content-Type:application/json" \
    -u devops:${ORDS_PASSWORD} \
    https://${ORDS_HOST}:8443/ords/_/db-api/stable/database/pdbs/${SRC_PDB_NAME}/
```

The use of environment variables maintained either by a Vault instance or locally by the CI Server increases code re-usability and security. The above example is very basic and does not make use of snapshot cloning functionality. Creating Copy-on-Write snapshots (not used in the example) can help reduce the storage footprint of the cloned database, provided your system uses compatible storage, and greatly reduce the overall cloning operation duration.

Please refer to the Oracle REST Data Services API documentation for more information about the PDB Lifecycle Management calls.

Using Recovery Manager's duplicate command

Recovery Manager (*RMAN*) is fully integrated with the Oracle Database to perform a range of backup and recovery activities, including the duplication of (pluggable) databases. Depending on your requirements you can use RMAN to duplicate the following sources:

- Non-container Databases (non-CDBs)
- Entire Container Databases (CDBs)
- Individual Pluggable Databases (PDBs) within a CDB

The integration of an RMAN duplicate step into a CI pipeline is straightforward. The desired RMAN commands should be saved to the Git repository as a script, ideally located in the top-level `utils` directory mentioned earlier in this tech brief.

Copyright © 2023, Oracle and/or its affiliates / Public

The main concern related to the use of RMAN's duplicate command is elapsed time. The larger the source, the longer it will take to complete the duplication. The time required to complete the duplicate command can be reduced by allocating an appropriate number of channels, the use of high-bandwidth network links and other means described in the Backup and Recovery User's Guide.

Using block-device Cloning Technology

Customers using traditional storage arrays on-premises and cloud customers using a block volume service can use block-volume cloning technology to quickly create copies of their databases. This option was used with non-CDB databases and is still available to customers using Oracle Database 19c with the traditional Oracle architecture. The process of block-volume cloning may require putting the database into backup mode to prevent in-flight I/O requests (that aren't necessarily sent to the database in order) from corrupting the copy.

Many storage vendors provide tools and procedures to clone block devices. It might be easiest to refer to these to automate the process, provided they offer an external API.

Using Copy-on-Write Technology

Copy-on-Write (COW) technology, also known as sparse clones, allows administrators to create full-sized copies of a database that only take a fraction of the space the source requires. Speaking greatly simplified, a sparse clone of a volume can be created quickly. From an operating system's point of view the sparse clone has the same properties as its source. Under the covers however, the storage software does not start copying every bit from the source to the target as it would with full clones. Sparse clones feature pointers to source data (the source volume). Only when data on the cloned volume changes, storage will be used. In other words, the amount of storage required for the cloned database is directly proportional to the amount of change.

This process can be highly beneficial in CI pipelines where typically 10% or less of the source database is changed. The potential downside of the approach – some overhead on the storage layer due to the maintenance of the delta – is typically compensated by the savings in storage, especially for larger databases.

COW technology predates the introduction of Oracle's Multitenant Option and thus can be used for 19c databases using non-CDB architecture as well as Pluggable Databases.

Not every storage engine and file system supports COW technology, please check with your storage vendor and Oracle if your solution supports COW cloning of (pluggable) databases.

Using Schema Provisioning

Schema provisioning marks the last, but not the least suitable mechanism for providing a deployment target. Schema provisioning is available for Multitenant and non-CDB environments alike.

In its most basic form, a user-created REST call creates a new schema in an existing Oracle database, returning the password to the CI pipeline. In the next step, the entire application must be provisioned. Just as with the new, empty Pluggable Database approach described earlier, this is potentially a time-consuming task.

It might be quicker to start off with a well-known/well-defined state – similar to the scenario described earlier in the context of cloning PDBs. Schemas in Oracle Database cannot be "cloned" using a SQL command. They can however be duplicated using Data Pump Export/ Data Pump Import. Assuming a suitable export file exists the CI pipeline can invoke Data Pump using a REST call. ORDS provides a set of REST endpoints allowing you to create a Data Pump Import Job.

Summary

Following the spirit of fast-feedback loops, CI pipelines must ensure that deployment targets are provisioned quickly. Customers who are already well into their cloud journey have lots of options at their disposal. Cloning Autonomous Database Serverless (ADB-S) instances ticks lots of boxes: starting from "testing on production-like data volumes" to "quickly provisioning the environment", few things are left to be desired. This approach might

Copyright © 2023, Oracle and/or its affiliates / Public

not be suitable in highly regulated environments where production data cannot be made available without proper data masking in place and many other compliance checks.

Cloning environments on-premises or in the cloud can be a suitable alternative for many customers. Sparse clones in particular offer a way out of the dilemma of having to provision TB worth of test environments.

Whichever approach you choose, please ensure that you run performance tests prior to go-live! Performance tests on production-like data are the only way to ensure the deployment pipeline does not run into timeouts due to long-running tasks.

Copyright © 2023, Oracle and/or its affiliates / Public

Writing effective CI/CD Pipelines

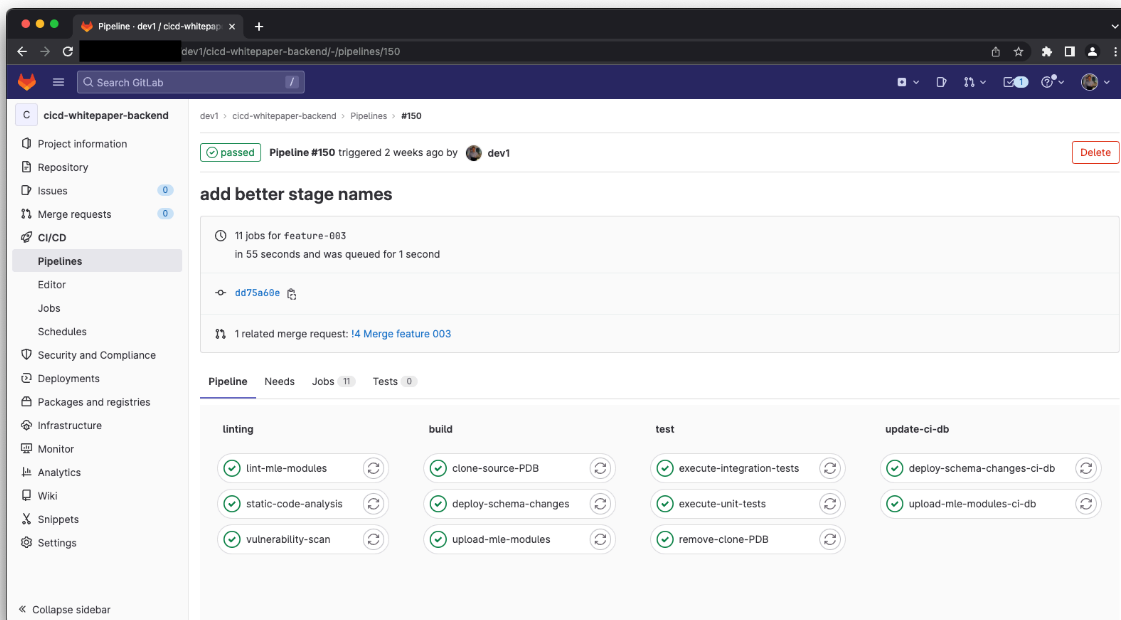
The previous chapters of this tech brief were intended to lay the foundation required for understanding how CI/CD pipelines can be defined. This chapter describes a hypothetical CI/CD pipeline based on GitLab Community Edition. Although the choice for this chapter fell to GitLab CE, the concepts described next apply to all CI servers from Jenkins to GitHub Actions. The use of GitLab is no endorsement of this technology.

Note: administration of CI/CD solutions like GitLab and GitHub can easily fill hundreds of pages. This chapter tries to cover the concepts and options necessary to get started with the given technology, it cannot be a replacement for the respective documentation.

Introduction to CI/CD Pipelines

The centerpiece of your automation project, CI/CD pipelines are typically defined in a markup language such as YAML. Some CI servers use their own domain-specific language. It is important for the pipeline's definition to be in a format that can easily be stored alongside the application code, as described in the earlier chapter concerning version control systems.

Figure 7. Example of a successful pipeline execution in GitLab

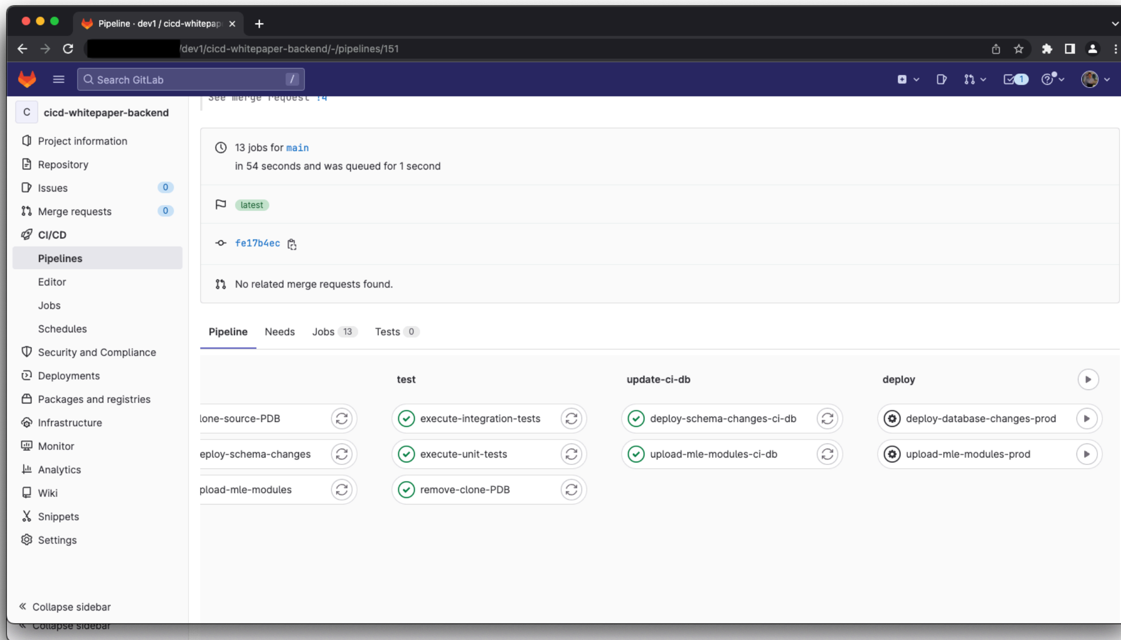


The above screenshot shows a successful pipeline execution. In this case this pipeline has been executed as part of a merge request (known as a pull request in GitHub). In GitLab, pipelines are run when the merge request is created and once more as part of the actual merge. Pushing a commit to GitLab is another option of triggering a CI pipeline's execution.

Depending on the activity you can start additional activities. Merging a branch is typically a more involved activity, especially when applying a **Trunk-Based Development** model. In that case, a merge to "trunk" or "main" can be used to deploy the application to the production environments. This is shown in the following figure:

Copyright © 2023, Oracle and/or its affiliates / Public

Figure 8. Example of a successful merge pipeline execution in GitLab



Unlike in Continuous Deployment where deployment is automated as well, the application is manually deployed to production in this example, visible by the "play" button in the deploy stage.

The screenshots have demonstrated some important concepts in the context of CI pipelines:

- Stages
- Jobs

CI Pipeline Stages

Stages allow you to group jobs. In the above example, linting, static code analysis and vulnerability scanning are performed in the linting stage. Stage names are completely arbitrary and can be chosen depending on the project's needs. Most CI servers allow the definition of stage names, like for example in Gitlab:

```
# stage definition
stages:
- linting
- build
- test
- update-ci-db
- deploy
```

Stages are typically completed in order. When designing stages you should consider the principle "fail early" literally: the sooner the pipeline fails, the quicker the developers can react. It is advisable to perform jobs requiring little to no time first, before starting on the ones that can take a while like cloning the source database.

CI Pipeline Jobs

Jobs are things the CI pipeline must perform, like passing a JavaScript module to typescript-eslint, jslint, jshint, and other linting tools. It could also involve cloning a "golden copy" PDB to create a deployment target. CI/CD pipelines group jobs logically into stages. Systems employing YAML syntax can define a job like this:

```
remove-clone-PDB:
  stage: test
  environment:
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
    name: testing
  script: |
    curl --fail -X DELETE \
    -d '{ "action": "INCLUDING" }' \
    -H "Content-Type:application/json" \
    -u devops:${ORDS_PASSWORD} \
   https://${ORDS_DEV}/ords/_/db-api/stable/database/pdbs/${CLONE_PDB_NAME}/
```

The `remove-clone-PDB` job is executed as part of the "`test`" stage and executes shell-script code (an HTTP POST request via the `curl` command line utility to the ORDS instance requesting the deletion of the cloned PDB after testing has completed).

Ensuring Code Quality

Code quality is one of the most important metrics when it comes to automating deployments. The State of DevOps Report regularly concludes that deploying frequently goes hand-in-hand with a lower failure rate. This might sound counter-intuitive, but thanks to code quality checks executed as part of the CI pipeline or pre-commit hooks this is a requirement that can be met.

Linting

According to Wikipedia, linting is a term used in computer science for a process where

- Programming errors …
- Many types of bugs …
- (Programming) Style …
- Other things …

… can be detected and/or enforced. Linting should occur as one of the first tasks during the execution of a CI pipeline. Code that doesn't adhere to the linting guidelines does not have to be deployed to find out that it will fail to work properly, the linting stage confirms that it is going to fail. A deployment therefore can be skipped and the pipeline's status set to "failure".

Errors during the linting phase should be rare: most **Integrated Development Environments** (**IDEs**) allow developers to include linters in the development process. Provided the developer's laptop uses the same linting rules and definitions as the pipeline, any potential errors should have been highlighted by the IDE and fixed prior to the commit.

Linting is no exact science, and one size doesn't fit all. Some rules the linter enforces by default might not be applicable to the project. In cases like this, the team usually decides which linting rules to use, and which to disable.

As with all other configuration settings, **Infrastructure as Code**, etc. the linter configuration should also be part of the Git repository.

Unit Testing

Once the code passes formal requirements it can be subjected to **Unit Tests**. There are many popular unit testing frameworks available. utPLSQL is useful for testing PL/SQL, and there are quite a few JavaScript unit testing frameworks like jest, mocha, and many more.

Using a JavaScript unit testing framework allows you to stay in the JavaScript ecosystem without having to learn a different unit testing framework. *Mocha* and *chai* prove to be a very popular combination of unit test tools. Depending on your preference you can run unit tests driven by the client-side node-oracledb database driver, or you can run the unit tests from within the database. The following example shows an excerpt of a unit test file using the client-side driver:

```
import { assert, expect, should } from "chai";
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
import { incrementCounter } from "../src/incrementCounter.mjs";
import oracledb from "oracledb";

describe("client-side unit test suite", function() {

    describe("ensure that a session is created", function() {

        it("should insert a row into the sessions table", async function () {
            const session_id = '123456C0340C0138E063020011AC3B29';
            // action
            incrementCounter(
                session_id,
                'node',
                'macOS'
            );

            // verification
            const conn = await oracledb.getConnection(
                {
                    user:             process.env.DB_USERNAME,
                    password:         process.env.DB_PASSWORD,
                    connectionString: process.env.DB_CONNECTIONSTRING
                }
            );

            const result = await conn.execute(
                `select
                    count(*)
                from
                    sessions
                where
                    session_id = :session_id `,
                {
                    session_id: {
                        dir: oracledb.BIND_IN,
                        val: session_id,
                        type: oracledb.STRING
                    }
                },
                {
                    outFormat: oracledb.OUT_FORMAT_ARRAY
                }
            );

            // there should only be 1 entry in the sessions table for this
            // particular session_id
            assert.strictEqual(
                result.rows[0][0],
                1,
                "there should only be 1 session per GUID"
            );
        });
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
    });
});
```

You can easily execute this test using your CI pipeline:

```
$ npx mocha

  client-side unit test suite
    ensure that a session is created
      □ should insert a row into the sessions table (220ms)

  1 passing (222ms)
```

Note: as with every third-party software you should always make sure to get consent from the relevant team to use it.

It is of course possible to use PL/SQL for unit testing. The following example demonstrates how to use utPLSQL, a very popular open source framework in this space, to run a unit test (the Liquibase-specific annotations have been left out for brevity). This example has been taken from a hypothetical application counting page hits.

```
create or replace package hit_counter_test_pkg as

    --%suite(unit tests for hit counter functionality)

    --%test(verify that a new session is created)
    --%tags(basic)
    procedure incrementCounter_test_001;

end hit_counter_test_pkg;
/

create or replace package body hit_counter_test_pkg as

    c_session_id constant varchar2(100) := 016050AA7B87051AE063020011ACAED8';
    c_browser           constant varchar2(100) := 'utplsql';
    c_operating_system  constant varchar2(100) := 'oracle';

    procedure incrementCounter_test_001 as
        l_cnt   pls_integer;
    begin
        -- Act
        hit_counter_pkg.increment_counter(
            p_session_id => c_session_id,
            p_browser    => c_browser,
            p_operating_system => c_operating_system
        );

        -- Assert
        select
            count(*)
        into
            l_cnt
        from
            hit_counts
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
        where
            session_id = c_session_id;

        -- there shouldn't be more than 1 rows inserted
        ut.expect( l_cnt ).to_equal( 1 );

    end incrementCounter_test_001;
end hit_counter_test_pkg;
/
```

There are many potential unit tests to be created, this article features a single test to demonstrate the point. Production applications should of course perform more thorough testing.

Performance Testing

Performance testing isn't necessarily initiated by a CI/CD pipeline due to the time-consuming nature of these tests; however, it is very important to conduct regular performance tests. These should ideally be targeted against a production workload. In the context of Oracle Database, there are a number of options available such as Real Application Testing or SQL Performance Analyzer. Depending on your license agreement these might be cost options.

Client-side load generators have also been successfully used to generate application load. Using these is another viable approach to performance testing as long as the usage characteristics of the real world can be represented as accurately as possible.

Deployment

Once the Integration part of the Continuous Integration pipeline has completed successfully it is time to deploy the change. Thanks to modern software development tools such as containers, deployment issues like library incompatibilities often encountered in the past are well addressed. Using a deployment pipeline to drive database changes using Liquibase, Flyway, or any other tool, the same can be achieved with database applications. Thanks to the meta-data preserved by these tools, scripts are guaranteed to be run only once. Therefore, it should be safe to define a main changelog referencing all changesets in the releases' `migrations` sub-directories of your project.

The question about the degree of deployment automation remains: Continuous Deployment in its pure form mandates that deployments are run against production as soon as they have passed all the tests defined in the pipeline. This however might not be risk-free and many departments are better off triggering the deployment manually. All major CI servers support a manual deployment clause in the pipeline. The following is an excerpt from a `.gitlab-ci.yml` file showing how to set the deployment step to "manual":

```
deploy-database-changes-to-prod:
  stage: deploy
  script:
  - cd src/database
  - |
    sql ${ORACLE_PROD_USER}/${ORACLE_PROD_PASSWORD}@${ORACLE_PROD_HOST}/${PROD_PDB_NAME} \
    @utils/deploy.sql ${TAG_NAME}
  environment:
    name: production
  when: manual
  rules:
  - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Thanks to the `when` attribute the execution of this step occurs manually.

Copyright © 2023, Oracle and/or its affiliates / Public

Deployments of your database changes are not limited to the CI database and production: any other tier should be considered equally important. The same deployment mechanism should be used for your User Acceptance Test, Integration Test and Performance Test environments. Most CI servers allow you to run pipelines manually passing variables to the process. These can be used to determine the destination servers.

Updating the CI database

Keeping the CI database up-to-date with the latest changes that have gone live into production is another important aspect of your CI/CD pipeline's execution. The CI database should always be as close to production as possible and hence changes that have been deployed into production need to be reflected in the CI database. Otherwise, future tests run at risk of failing due to these missing changes, or worse, trying to repeat the changes that are already present in production.

Thanks to tools like Liquibase or Flyway, all changes that are **not yet applied** to a database (the **change delta**) will be rolled out automatically. This is not only true for deployments into production, but also for the CI databases that are provisioned as part of the test execution. However, the larger the change delta gets, i.e., the more changes that the CI database lags behind production, the longer the CI database setup phase will take, eventually slowing down your overall pipeline execution duration.

To remedy that, CI databases should regularly be updated with the latest changes so that the change delta stays at a minimum.

Based on your deployment frequency and whether you employ Continuous Delivery or Continuous Deployment, you will have to decide when and how to update the CI database.

If you plan on intra-day deployments straight to production, you may want to make updating the CI database with the changes part of your deployment stage.

If you, on the other hand, employ Continuous Delivery or have infrequent deployments, a regular refresh of the "gold image" from production may be enough.

Summary

Creating effective CI/CD pipelines for a software project is very rewarding once all the project's requirements have been implemented. The task however isn't trivial, sufficient time should be set aside to plan and create the pipeline. The amount of coordination required between teams and the potential change in culture should not be underestimated. It is advisable to provide some time as a contingency in the project's planning stage.

CI/CD pipelines are typically written in YAML or comparable markup languages. As with any other application artefact they should be part of your project's Git repository.

The golden rule with CI/CD pipelines is to "keep the pipeline green", in other words, not to introduce issues. Developers can make use of local tests (linting, unit tests, code coverage, etc.) before pushing a commit to the remote repository. **Test-Driven Development** (**TDD**), combined with **Trunk-Based Development** has proven a successful combination, as visible in many State of DevOps reports.

Copyright © 2023, Oracle and/or its affiliates / Public

Performing Schema Changes Online

The previous chapters provided an overview of how to deploy schema changes effectively. Combining CI/CD pipelines and a development workflow that's right for your team enables you to deploy small, incremental changes to the application with a high degree of confidence that they won't break production.

Deploying to Production with Confidence

Deployments to production are special: extra care must be taken not to interrupt ongoing operations. For many systems stopping production workloads to deploy a software release has been impossible for many years, and such drastic measures shouldn't be required anymore.

One of the concerns voiced by developers is related to the (misperceived) inability of relational databases to perform online schema migrations. This chapter aims to address these concerns, demonstrating that application changes can indeed be performed online.

Avoiding outages, however brief they might be

Schema migrations with Oracle Database are different from many other relational database management systems (RDBMS). The Oracle Database has been able to perform many **DDL** (**Data Definition Language**) operations online for decades. Online index rebuilds, for example, have been available from as early as Oracle 8i. Adding columns to tables, rebuilding indexes, or even code changes in PL/SQL don't have to result in extended periods of locking database objects and blocking workloads to run.

Please note that the Oracle Database offers far more online operations than covered in this chapter with its focus on application development. Please refer to the Oracle Database Development Guide, Database Concepts, and the Database Administrator's Guide for a complete picture.

Oracle has an entire team dedicated to designing a **Maximum Availability Architecture (MAA)**. Their work is very important when it comes to maintaining the underlying infrastructure during planned and unplanned outages. There is a certain overlap with this tech brief, you are encouraged to review the MAA tech briefs in addition to this one.

Online operations

The difference in elapsed time between online operations and blocking operations is striking, especially if the objects to be changed are frequently accessed. Oracle guarantees that structural changes to a schema object like a table, partition, or index cannot be applied while a transaction changing the contents of the segment is active. The same is true for a piece of business logic being executed, such as a trigger, or stored. The rationale is to ensure consistency as well as integrity. This is both intended and a good principle.

Online operations in the context of this chapter refer to those operations that have been optimised to require locks only for the shortest period of time, if at all. They are easy to spot as they typically add the `ONLINE` keyword to the DDL command. Some features discussed in this chapter might require an extra license, always consult the Database Licensing Guide when in doubt.

The Oracle SQL Language Reference contains a list of non-blocking DDL operations per release.

Creating Indexes Online

Index creation and index rebuilding have been part of the Oracle Database engine for more than 25 years. Introduced in the 8i timeframe, developers use the `ONLINE` keyword to indicate that regular DML operations on the table will be allowed during the creation of the index.

The index creation cannot be performed entirely without a brief period of locking, however, that time should be very short. The index creation or rebuilding command will queue for its lock, just like any ongoing transaction does.

Copyright © 2023, Oracle and/or its affiliates / Public

Initially, the foreign key referencing SESSION_ID in table HIT_COUNTS was unindexed. The following script adds the index online.

```
--liquibase formatted sql
--changeset developer1:"r2-02" failOnError:true labels:r2

CREATE INDEX i_hit_counts_sessions ON
    hit_counts (
        session_id
    )
        ONLINE;
```

Using the above script, the index was added to the application without interrupting users from performing DML operations against the table.

Introducing partitioning to an existing, non-partitioned table

Despite the best planning efforts sometimes unexpectedly high data volume makes it necessary to enable partitioning for a table that was previously unpartitioned.

Oracle offers multiple technologies to introduce partitioning for tables: the below ALTER TABLE command as well as the DBMS_REDEFINITION PL/SQL package. The latter serves additional use cases and will be covered in more detail later.

A requirement to preserve entries in the application's HIT_COUNTS table mandates partitioning the table by range based on the HIT_TIME column. The following SQL command performs this operation online. At the same time, the previously added index is converted to a locally partitioned index.

```
--liquibase formatted sql
--changeset developer1:"r2-03" failOnError:true labels:r2
ALTER TABLE hit_counts MODIFY
    PARTITION BY RANGE (
        hit_time
    ) INTERVAL
       ( numtoyminterval(
           1, 'MONTH'
       ) )
    ( PARTITION p1
        VALUES LESS THAN ( TO_TIMESTAMP('01.01.2000', 'dd.mm.yyyy') )
    )
    ONLINE
    UPDATE INDEXES (
        i_hit_count_session LOCAL
    );
```

The above example introduces interval partitioning to the table based on the HIT_TIME timestamp. Data is automatically sorted into the correct partition thanks to the NUMTOYMINTERVAL() function. This is merely one example of the possibilities you have available: you can change the partitioning scheme in almost any way you want, including indexes.

Compressing a segment online

Both tables and table (sub-) partitions can be compressed online. Following the previous example of introducing range partitioning to HIT_COUNTS you can compress the oldest segment online:

```
--liquibase formatted sql
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
--changeset developer1:"r2-04" failOnError:true labels:r2
ALTER TABLE hit_counts
MOVE PARTITION p1
COMPRESS BASIC
ONLINE;
```

Partition P1 is now compressed using `BASIC` compression. Depending on your platform, you might be able to achieve better compression levels using **Advanced Compression Option** or **Hybrid Columnar Compression** (**HCC**).

If you don't want to own the process of compressing older, read-only data you may be interested in **Automatic Data Optimization** (**ADO**). It uses a heat map to record segment activity and allows you to define **Information Lifecycle Management** (**ILM**) policies such as moving segments to different tablespaces, and/or compressing them as part of the policy execution.

Adding Columns to Tables

Adding columns to tables is a typical task for any developer. Oracle Database optimised the process of adding new columns. *Nullable* columns *without default* value can be added to the table without any interruption since Oracle Database 11.2. Likewise, columns that are defined as NOT NULL can be added online when using default values. Prior to Oracle Database 11.2 adding a NOT NULL column with a default value required an update of the entire table to store the default value in the column after the column is added, causing significant load on the storage system and other overhead. Oracle Database 11.2 changed this to a metadata-only operation, breaking the correlation between the elapsed time to execute the command and the size of the table. Oracle Database 12.1 added support for nullable columns with a default value as well, transforming the addition of columns to tables into an online operation.

The completion of the `ALTER TABLE … ADD COLUMN` command will wait until all previous transactions prior to the `ALTER TABLE … ADD COLUMN` command are finished, then briefly lock the table and finish. In a sense, the `ALTER TABLE … ADD COLUMN` command is just like any other transaction that is queued until all previous transaction holding locks are finished. Note that there is no dedicated `ONLINE` keyword.

Using Online Table Redefinition to Change Table Structures Online

Oracle Database provides a mechanism to allow users to make table structure modifications without significantly affecting the availability of the table to other users and workloads. The mechanism is called online table redefinition and is exposed via the `DBMS_REDEFINITION` PL/SQL package. Redefining tables online provides a substantial increase in availability compared to traditional methods of redefining tables manually.

When a table is redefined online, it is accessible to both queries and DML operations during much of the redefinition process. Typically, the table is locked in exclusive mode only during a very small window that is independent of the size of the table and the complexity of the redefinition, and that is completely transparent to users. However, if there are many concurrent DML operations during redefinition, then a longer wait might be necessary before the table can be locked.

The application's `SESSIONS` table is defined as a relational table featuring 4 columns:

- SESSION_ID – the GUID provided by the frontend.
- BROWSER – the browser invoking the application.
- OPERATING_SYSTEM – the client's operating system.
- DURATION – the duration of the session.

Copyright © 2023, Oracle and/or its affiliates / Public

The team decided to change the table structure, combining the latter 3 columns into a JSON document. The change should be performed while the application remains online. It is good practice to check if the source table can be redefined:

```
begin
  sys.dbms_redefinition.can_redef_table ('DEMOUSER', 'SESSIONS');
end;
/
```

If no errors or exceptions are thrown by the above procedure, the process can be initiated. First, you define a target table. The target table defines how the table you are redefining should look like once the process is finished. The documentation refers to it as the *interim table*:

```
CREATE TABLE sessions_json (
    -- this is a UUID
    session_id          char(32) NOT NULL,
    session_data        json
);
```

Note that existing indexes on the source table will be copied automatically as part of the procedure to the interim table, therefore there is no primary key/unique index defined on the interim table. Very large tables might benefit from parallel DML and parallel query. Enable as necessary, provided your workload allows it and you have sufficient resources available on your database server to handle the extra load.

Now the process can be started:

```
BEGIN
 DBMS_REDEFINITION.start_redef_table(
   uname        => 'DEMOUSER',
   orig_table   => 'SESSIONS',
   int_table    => 'SESSIONS_JSON',
   col_mapping =>
     'session_id session_id, ' ||
     'json_object(browser,operating_system,duration returning json) session_data',
   options_flag => DBMS_REDEFINITION.cons_use_pk
  );
END;
/
```

`COL_MAPPING` is by far the most important parameter in this code snippet. Using the column mapping string, you define how to map columns between the source and interim tables. The first parameter is an expression, the second parameter denotes the interim table's column name. In this example

- `SESSION_ID` is mapped to `SESSION_ID` – no change, the column names and data types are identical.
- A call to `JSON_OBJECT()` featuring the relational columns from the source table is mapped to the `SESSION_DATA` column in the interim table.

It is possible to copy the table dependents over as well. The example below copies the unique index and privileges to the interim table but not the primary key constraint or triggers. Note that statistics must be gathered manually after the redefinition operation as well, since `COPY_STATISTICS` is `FALSE`, a conscious decision given that the table structures of the source and interim tables are different and hence statistics for the new column in the interim table do not exist in the source table. All the other parameters are set based on the application's needs. Any errors encountered while executing the code block will raise an exception thanks to `IGNORE_ERRORS = FALSE`.

```
DECLARE
  l_errors PLS_INTEGER;
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
BEGIN
  DBMS_REDEFINITION.copy_table_dependents(
    uname                    => 'DEMOUSER',
    orig_table               => 'SESSIONS',
    int_table                => 'SESSIONS_JSON',
    copy_indexes             => DBMS_REDEFINITION.cons_orig_params,
    copy_triggers            => FALSE,
    copy_constraints         => FALSE,
    copy_privileges          => TRUE,
    ignore_errors            => FALSE,
    num_errors               => l_errors,
    copy_statistics          => FALSE
  );
  dbms_output.put_line('error count: ' || l_errors);

  if l_errors != 0 then
    raise_application_error(
      -20001,
      l_errors || ' encountered trying to copy table dependents'
    );
  end if;
END;
/
```

Once the command completes it is time to add the constraints. In the above case there's only one to be added: the primary key.

```
ALTER TABLE sessions_json
  ADD CONSTRAINT pk_sessions_json
  PRIMARY KEY ( session_id );
```

During the redefinition process, you can synchronize the interim table data with the source table. After the redefinition process has been started by calling `START_REDEF_TABLE` and before it ended by calling `FINISH_REDEF_TABLE`, a large number of DML statements may have been executed on the source table. If you know that this is the case, then it is recommended that you periodically synchronize the interim table with the source table. There is no limit to how many times you can call `SYNC_INTERIM_TABLE()`.

```
BEGIN
  DBMS_REDEFINITION.sync_interim_table(
    uname                 => 'DEMOUSER',
    orig_table            => 'SESSIONS',
    int_table             => 'SESSIONS_JSON',
    continue_after_errors => false
  );
END;
/
```

The interim table can be queried by your development team to ensure that the structure, contents, and any other properties of importance match your expectations. If so, you can finish the redefinition process:

```
BEGIN
  DBMS_REDEFINITION.finish_redef_table(
    uname                 => 'DEMOUSER',
    orig_table            => 'SESSIONS',
```


Copyright © 2023, Oracle and/or its affiliates / Public

```
    int_table             => 'SESSIONS_JSON',
    dml_lock_timeout      => 60,
    continue_after_errors => false
  );
END;
/
```

As soon as the prompt returns you should gather statistics on the table.

```
BEGIN
  -- table prefs define all the necessary attributes for stats gathering
  -- they are not shown here
  DBMS_STATS.gather_table_stats('DEMOUSER', 'SESSIONS');
END;
/
```

This concludes the table redefinition example.

Next-level Availability: Edition-Based Redefinition

Edition-based redefinition (**EBR**) enables online application upgrades with uninterrupted availability of the application by versioning application code and data model structures using editions. When the rollout of an application upgrade is complete, the pre-upgrade version of the application and the post-upgrade version can both be in active usage at the same time.

Using this mechanism, existing sessions can continue to use the pre-upgrade application version until usage of it reaches its natural end; and all new sessions can use the post-upgrade application version. When there are no more sessions using the pre-upgrade application version, the version can be retired.

In this way, EBR allows hot rollover from the pre-upgrade version to the post-upgrade version, with zero downtime.

Adopting EBR can happen in multiple steps, and it is perfectly fine not to progress toward the final level described in the following sections. Anything that helps making your application more resilient to changes is a win!

EBR Concepts

As the name implies, **Edition-based redefinition** is based around editions. Editions are non-schema objects; as such, they do not have owners nor reside inside any schema. Editions are created in a single namespace, and multiple editions can coexist in the database. Editions provide the necessary isolation to re-define schema objects of your application.

Database objects such as packages, procedures, triggers and views can all be editioned. Any application using any such objects as part of its execution can leverage EBR to introduce a changed object under a new edition (or version) without changing or removing the current object. You as the user or the application itself can then decide when to use which edition and the database will resolve the correct version of the objects accordingly.

Tables are not editioned, and they cannot be. You work with *editioning views* instead in cases where changes to tables are required. Note that the creation of an editioning view requires an application outage, but that might as well be the last outage you must take if you fully embrace EBR. On an editioning view, you can define triggers just like on a table, with the exception of crossedition triggers, which are temporary, and `INSTEAD OF` triggers. Therefore, and because they can be editioned, editioning views let you treat their base tables as if the base tables were editioned.

In the scenario where other users must be able to change data in the tables while you are changing their structure, you can use forward crossedition triggers to also store data in the new structure elements, e.g. a new column or column with changed data type, etc.. If the pre- and post-upgrade applications will be in use at the same time (hot rollover), then you can also use reverse crossedition triggers to store data from the post-upgrade application in

Copyright © 2023, Oracle and/or its affiliates / Public

the old structure elements for the pre-upgrade application to consume. Crossedition triggers are not a permanent part of the application—you drop them once all users are using the post-upgrade application version. The most important difference between crossedition triggers and noncrossedition triggers is how they interact with editions.

Adoption Levels

EBR is incredibly powerful, but with its power comes a certain complexity using it. Thankfully, EBR can be adopted in levels, each additional layer providing more resilience to application changes.

Level 1

The first adoption level aims at enabling changes to the backend PL/SQL APIs without incurring library cache locks.

Objects in the library cache, such as PL/SQL code, are protected from modifications via DDL locks. A PL/SQL stored procedure currently in use cannot be replaced until all sessions have finished using it. Replacing core PL/SQL functionality of a busy system can be very hard without quiescing the database. EBR makes this a lot easier.

All other changes are implemented without the help of EBR features.

An example of this process is documented in the Database Development Guide, section 32.7.2.

Level 2

The next level of EBR adoption allows developers to implement PL/SQL changes in a new edition just like with level 1. Additionally, they use editioning views. The current version of the application is not affected, in other words, your application code does not require cross-edition data access and tables being redefined are not accessed by users during the application maintenance operation.

All other changes are implemented without the help of EBR features.

You can see an example of this process in the Database Development Guide, section 37.7.3.

Level 3

Adoption of level 3 implies the use of all previous levels, except that it is necessary to transfer data between editions. In a select few cases, cross-edition triggers are in use where the effort to implement them is low. Only the busiest tables are cross-edition enabled.

All other changes are implemented without the help of EBR features.

You can see an example of this process in the Database Development Guide, section 37.7.4.

Level 4

Users of adoption level 4 perform all their application changes using every EBR feature available. Potentially they never require outages to perform application changes, however, this milestone requires a solid investment into the technology and the automation of change management processes. The difference between level 3 and 4 is the scope: with level 3 adoption only a select few tables are cross-edition enabled whereas with level 4 every table is.

You can see an example of this process in the Database Development Guide, section 37.7.4.

Potential Workflows

Before you can use EBR to upgrade your application online, you must prepare it first:

1. Editions-enable the appropriate database users and the appropriate schema object types in their schemas.
2. Prepare your application to use editioning views. An application that uses one or more tables must cover each table with an editioning view.

The following steps represent a possible workflow for deploying application changes using EBR:

Copyright © 2023, Oracle and/or its affiliates / Public

1. Create a new edition.
2. Alter your session to use the newly created edition.
3. Deploy application changes.
4. Ensure that all objects are valid.
5. Perform unit testing and integration testing.
6. Make the new edition available to all users and make it the default.

Services should be used to connect to the Oracle Database, but not all services are equal. The auto-generated service name, for example, is to be used for database administration only. All applications should connect to their dedicated service created as part of the application's initial deployment. Once the rollout of the new edition is completed you can change the edition property of the service to point to the new edition.

Summary

Oracle Database has a proven track record of performing schema migrations online. Adding columns to tables, creating indexes, introducing partitioning as well as partition maintenance operations can be executed without introducing an unnecessary burden on the application's uptime.

Edition-based redefinition, a feature exclusive to Oracle Database, can be used for hot deployments of application changes. Its adoption does not require a big-bang approach, it can be retrofitted using a staggered approach up to the degree you are comfortable with. Even if you decide to only manage PL/SQL changes online there are potentially huge gains to be had compared to the standard approach. Additionally, if your application uses APIs written in PL/SQL to decouple the frontend from the database backend, you can break the link between their respective release cycles.

Copyright © 2023, Oracle and/or its affiliates / Public

ORACLE

Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

blogs.oracle.com

facebook.com/oracle

twitter.com/oracle

Copyright © 2023, Oracle and/or its affiliates. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Author: Martin Bach, Senior Principal Product Manager, Oracle

Contributors: Ludovico Caldara, Senior Principal Product Manager, Oracle; Connor McDonald, Developer Advocate, Oracle; Gerald Venzl, Senior Director, Oracle