Implementing DevOps principles with Oracle Database

September, 2023, Version 1.0
Copyright © 2023, Oracle and/or its affiliates
Public
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 manager 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.
# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
<td>6</td>
</tr>
<tr>
<td>Towards a Modern Software Development Workflow</td>
<td>6</td>
</tr>
<tr>
<td>Implementing Continuous Integration</td>
<td>7</td>
</tr>
<tr>
<td>Continuous Delivery and Deployment</td>
<td>9</td>
</tr>
<tr>
<td>Attributes of Continuous Integration and Delivery</td>
<td>9</td>
</tr>
<tr>
<td>Version Control</td>
<td>9</td>
</tr>
<tr>
<td>Fast Feedback Loops</td>
<td>10</td>
</tr>
<tr>
<td>Automated Testing</td>
<td>10</td>
</tr>
<tr>
<td>Small and Frequent Changes</td>
<td>10</td>
</tr>
<tr>
<td>Automated Deployment</td>
<td>10</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>11</td>
</tr>
<tr>
<td><strong>Using Version Control Systems</strong></td>
<td>12</td>
</tr>
<tr>
<td>No CI/CD without the use of a version control system!</td>
<td>12</td>
</tr>
<tr>
<td>Getting the team’s buy-in to using a Version-Control-System</td>
<td>12</td>
</tr>
<tr>
<td>Introduction to Git</td>
<td>12</td>
</tr>
<tr>
<td>Git Terminology</td>
<td>13</td>
</tr>
<tr>
<td>Branching and Merging</td>
<td>13</td>
</tr>
<tr>
<td>Pull and Merge Requests</td>
<td>13</td>
</tr>
<tr>
<td>Forking a Project</td>
<td>14</td>
</tr>
<tr>
<td>Your Database Project’s Git Repository</td>
<td>14</td>
</tr>
<tr>
<td>Release Management</td>
<td>16</td>
</tr>
<tr>
<td>Integrating with the Continuous Integration Pipeline</td>
<td>16</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>Ensuring Repeatable, Idempotent Schema Migrations</strong></td>
<td>17</td>
</tr>
<tr>
<td>Benefits of using Dedicated Schema Migration Tools</td>
<td>17</td>
</tr>
<tr>
<td>Using SQLcl and Liquibase to deploy Schema Migrations</td>
<td>18</td>
</tr>
<tr>
<td>Liquibase Terminology and Basic Concepts</td>
<td>18</td>
</tr>
<tr>
<td>Practical Aspects Creating the Database Changelog</td>
<td>19</td>
</tr>
<tr>
<td>Formatting your changelog</td>
<td>19</td>
</tr>
<tr>
<td>Deploying the Changelog</td>
<td>20</td>
</tr>
<tr>
<td>Checking the status of your deployments</td>
<td>21</td>
</tr>
<tr>
<td>Ensuring Pipeline Failure</td>
<td>23</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>24</td>
</tr>
<tr>
<td><strong>Efficient and Quick Provisioning of Test Databases</strong></td>
<td>25</td>
</tr>
<tr>
<td>Autonomous Database-Serverless</td>
<td>25</td>
</tr>
<tr>
<td>Using Container Images</td>
<td>26</td>
</tr>
<tr>
<td>Using container images with Podman or Docker</td>
<td>27</td>
</tr>
<tr>
<td>Using Container Images with Kubernetes</td>
<td>28</td>
</tr>
<tr>
<td>Using Container Databases</td>
<td>29</td>
</tr>
<tr>
<td>Creating a new, empty Pluggable Database</td>
<td>29</td>
</tr>
</tbody>
</table>
Cloning an existing Pluggable Database  29
Automating Pluggable Database Lifecycle Management  30
Using block-device Cloning Technology  30
Using Copy-on-Write Technology  30
Using Schema Provisioning  31
Summary  31

**Writing effective CI/CD Pipelines**  32
Introduction to CI/CD Pipelines  32
CI Pipeline Stages  33
CI Pipeline Jobs  33
Ensuring Code Quality  34
  Linting  34
  Unit Testing  34
  Performance Testing  37
Deployment  37
Updating the CI database  38
Summary  38

**Performing Schema Changes Online**  39
Deploying to Production with Confidence  39
Avoiding outages, however brief they might be  39
Online operations  39
Creating Indexes Online  39
Introducing partitioning to an existing, non-partitioned table  40
Compressing a segment online  40
Adding Columns to Tables  41
Using DBMS_REDEFINITION to redefine Tables Online  41
Next-level Availability: Edition-Based Redefinition  43
  EBR Concepts  44
  Adoption Levels  44
  Potential Workflows  45
Summary  45
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 32
Figure 8. Example of a successful merge pipeline execution in GitLab 33
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 frontend developers have been able to take huge strides in automating and bundling software releases for many years now, database applications have not enjoyed the same level of attention and automation. Many frontend 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 functionality provided by the database within the application.
- Duplication of database-provided 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 against 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.
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) 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 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):
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 artifactory”. 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 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:

```yaml
# ----------------------------------------------------------- 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
```
# 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 Test (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 you can’t 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 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 being made to production.

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 waiting weeks before difficulties are detected it might be easier to combine changes in a branch 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 code quality. DevOps engineers must therefore ensure the pipeline’s runtime remains short. This can be challenging especially when it comes to providing a full copy of the test database to the pipeline.

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

**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 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 problems in code tends to be proportional to the size of the change. A feature branch that hasn’t been merged back into the main or whatever your release branch is for weeks carries a larger risk of creating merge conflicts than a small change that’s been merged within a few hours up to a day.

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 that 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.

**Automated Deployment**

In a database project there is no single build artifact as in a Java project for example. The application isn’t built to 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 the project to be a success the deployment mechanism must be identical no matter what environment you are deploying against. The deployment must 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 previously undetected errors. In environments where a lot of administrative work is manually performed there is a high risk that a deployment to a development environment has almost no resemblance to UAT, Integration, or worse: production. 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 and database clones don’t have to take enormous amounts of time.
Chapter 3 discusses how to create repeatable, idempotent deployments of database schema migrations.

**Summary**

Many database-centric application projects are handled differently compared to 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.
Using Version Control Systems

This section covers version control systems (VCS), particularly Git, the most common version control system at the time of writing.

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.

No CI/CD without the use of 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 the Git repository. That can imply greater visibility of an individual’s contributions, something not everyone is comfortable with. Project members reluctant to work with Git can be convinced by pointing out the benefits of Git, 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 Git. 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 Git. You could even go so far as to appoint a “Git 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-
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.

*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 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 clone/contribute to the team’s members.

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 main 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. All files in the staging area 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. Delay however 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 (found on 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 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, and various other workflow details. Most importantly, all the commits from the source branch and all the files changed are listed, allowing everyone to assess the impact and comment on the code.

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 in 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 or deleted.

**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 Repository**

There is an ongoing discussion in the developer community about whether application code like that of your Angular, React, or any other frontend should coexist with database changes 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 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 silo’d 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 status 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.

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, 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
    ├── newRelease.sh
    │   └── r1
    │       ├── migrations
    │       │   └── state
    │       ├── testdata
    │       └── utils
    │   └── r2
    │       ├── migrations
    │       │   └── state
    │       ├── testdata
    │       └── utils
    └── r3
        ├── migrations
        │   └── state
        ├── testdata
        └── utils
```

One directory exists per 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.

Database changes are stored in the migrations folder, and in the case of Liquibase deployments they are executed in alphabetical order if they aren't named explicitly.

The state directory contains the schema DDL for all objects exactly in the way they should look like at the end of the successful schema migration. This directory can be used as reference and for troubleshooting. It should never be used for code 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 steps, never is such a script allowed to perform changes to the database schema.
As soon as a 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.

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 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 n-1 to n; you can also start earlier, for example, at n-5. Thanks to the aforementioned schema migration tools, only those scripts that haven’t been applied yet will be applied as part of the 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. Using the DDL statement 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.

It is very important only to 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 split your code in separate repositories for database and frontend 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.
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 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 pure frontend 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. The database part of the project can be found in ./src/database. 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
  │   ├── migrations
  │   │   ├── state
  │   │   └── testdata
  │   └── utils
  ├── newRelease.sh
  │   ├── migrations
  │   │   ├── state
  │   │   └── testdata
  │   └── utils
  └── r1
      ├── migrations
      │   ├── state
      │   └── testdata
      └── utils

r2
  ├── migrations
  │   ├── state
  │   └── testdata
  └── utils

r3
  ├── migrations
  │   ├── state
  │   └── testdata
  └── utils
```

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:
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. You can leverage all the great features built into SQLcl to connect to an Oracle Database. Once connected to the target environment it is possible to deploy changes using 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 set.

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.
Changesets are the basic entities developers create as part of the release. Multiple changesets are referred to as your database’s changelog. A changeset contains 1 or more change types. As a best practice limit yourself to 1 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 scripts with Liquibase-specific annotations.

**Practical Aspects 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 changesets:

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

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 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 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:

```sql
--liquibase formatted sql
```
As you can see in the example, the SQL file must feature the Liquibase preamble followed by the changeset information (author:ID, e.g. developer1:"r1-01"). Additional flags 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 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 change sets can be written along the same lines, albeit additional attributes 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 "\n" character in the examples just indicates that the text continues on the same line and has only been shown in another line due to the limited page width.)

--liquibase formatted sql
--changeset dev2:"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 the label like in the previous example. stripComments is set to false to ensure none of the comments inside the PL/SQL code are removed, which would break the unit test. The SQL command is defined to terminate with the / character, as instructed by the endDelimiter attribute.

**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 flag is set, which is not recommended).
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
<?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 release (rN) is added using a new includeAll directive. All the Liquibase changelogs 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 rN/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. The main changelog references r2/migrations as the path attribute of the includeAll tag.

Thanks to the metadata checks, any changeset found in the baseline and migrations folder that has already been executed is skipped; no conditional logic is required. Since the main changelog 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. There are 2 potential strategies to deal with this situation:

- Forward fixing: a follow-up release corrects the mistake, provided that is possible.
- Rollback: either using database mechanisms (Flashback database, database restore/recovery, etc.) or Liquibase’s rollback mechanism.

Please note that Liquibase’s own rollback mechanism is not subject of this tech brief. Ideally, any issues with a changeset are identified prior to its deployment! Catching errors early in the CI pipeline is covered in a later section.

### 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

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=1521))(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

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. Auditing of 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 audit. 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 flags 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. Just like SQL*Plus, users can specify the `whenever sqlerror exit` command.

Many users chose to create their own deployment script in the `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:

```sql
/*
 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_
 */

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(
    '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;
/

whenever sqlerror exit
create restore point t_&1;
lb tag -tag t_&1
lb update -changelog-file controller.xml -debug true

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 doesn't 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.
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 section 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 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 (Shared) Database.
- 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-Serverless

Oracle Autonomous Database provides an easy-to-use, fully autonomous database that scales elastically and delivers fast query performance. As a 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 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:

```terraform
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), well integrated with the CI Server and the pipeline's infrastructure and Network Security Group (NSG). This is 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-Serverless 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 can be deployed everywhere, from a developer's laptop all the way up the tiers into production. They 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.
Figure 5. Screenshot showing Database container images on Oracle’s container registry

<table>
<thead>
<tr>
<th>Repository</th>
<th>Description</th>
<th>Open Source License Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>express</td>
<td>Oracle Database Express Edition</td>
<td>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.</td>
</tr>
<tr>
<td>free</td>
<td>Oracle Database Free</td>
<td>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.</td>
</tr>
<tr>
<td>observability-exporter</td>
<td>Oracle Database Observability Exporter (Metrics, Logs, and Tracing)</td>
<td>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.</td>
</tr>
<tr>
<td>operator</td>
<td>This image is part of and for use with the Oracle Database Operator for Kubernetes</td>
<td>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.</td>
</tr>
<tr>
<td>ords</td>
<td>Oracle REST Data Services (ORDS) with Application Express</td>
<td>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.</td>
</tr>
<tr>
<td>otm/m</td>
<td>Oracle Transaction Manager for Microservice</td>
<td>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.</td>
</tr>
<tr>
<td>sqlcl</td>
<td>Oracle SQLDeveloper Command Line (SQLcl)</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

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

27 Implementing DevOps principles with Oracle Database / Version 1.0
Copyright © 2023, Oracle and/or its affiliates / Public
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 passwords to the value stored as a Podman secret named oracle_pwd. After less than 1 minute the database is started and ready. It 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, read: less time-consuming approach.

### Using Container Images with Kubernetes

Advanced users of container technology might want to deploy the database container in Kubernetes or a comparable orchestration engine. Instead of manually provisioning storage and other Kubernetes cluster resources administrators can make use of 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 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 supports a multitude of database configurations and infrastructure including support for Autonomous Database on Shared Exadata Infrastructure. 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:

```yaml
apiVersion: database.oracle.com/v1alpha1
kind: SingleInstanceDatabase
metadata:
  name: freedb-sample
  namespace: default
spec:
  ## Use only alphanumeric characters for sid
  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 Express Edition/Oracle Free.
  replicas: 1
```

# 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.

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 PDBs. A PDB is a user-created set of schemas, objects, and related structures that appears 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

Additonal 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 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. 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.

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 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.

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

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 previously used with non-CDB databases and is still available to customers using Oracle 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 takes a fraction of the space the source requires. Speaking greatly simplified sparse clones look the same as their source to Oracle, however, under the covers the storage engine merely allocates space but doesn't copy the source to the destination. Pointers in the target database's storage engine point to the original data, and only when something on the target side changes is the pointer replaced with the changed information. 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 less than 10% - or less – of the source database is changed. The potential downside of the approach – some overhead 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 beforehand with your storage vendor 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 option 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 Export Data Pump/Import Data Pump. 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 an Import Data Pump Job.

**Summary**

Following the spirit of fast-feedback loops CI pipelines must ensure that deployment targets are provisioned quickly. Customers already well in to their cloud journey have lots of options at their disposal. Cloning Autonomous Database on Shared Exadata Infrastructure (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 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.
**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:
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:

```yaml
# --------------------------------------------- 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:

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

33 Implementing DevOps principles with Oracle Database / Version 1.0
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 (a HTTP POST request against the ORDS instances 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](https://en.wikipedia.org/wiki/Linting), 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 is 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 in the editor, 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 test 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:

```javascript
import { assert, expect, should } from "chai";
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"
    );
  });
});
You can easily execute this test using your CI pipeline:

```bash
$ 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 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.

```sql
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
    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 Analyser. 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 another 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
    - |
      
      {"whenever sqlerror exit;"}
      echo "lb tag -tag $CI_COMMIT_SHORT_SHA"
      echo "lb update -changelog-file controller.xml -log -debug"
      echo "exit"
      } | sql {ORACLE_PROD_USER}/${ORACLE_PROD_PWD}@"${ORACLE_PROD_CONN_STRING}"

  environment: production
  when: manual
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

Thanks to the “when” attribute the execution of this step occurs manually.
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 is another important aspect of your CI/CD pipeline’s execution. Depending on your change ratio the CI database might lag behind, potentially introducing unnecessary delay. If you decide to apply changes that went live to the “golden image” copy, you can speed up the CI part of your pipeline’s execution.

In case you don’t plan on intra-day deployments it might be equally effective to perform a nightly refresh of the “golden image” from production.

**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 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 DORA-State of DevOps reports.
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 you can 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.

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 Developer’s 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 between online operations and blocking operations is striking, especially if the objects to be changed are frequently accessed must be changed. 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 such as a trigger, or stored procedure is executed. The rationale is to ensure consistency as well as integrity. This is both intended and a Good Thing.

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 a command. Some features discussed in this chapter might require an extra license on top of Enterprise Edition. Always consult the Licensing Guide when in doubt.

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.

Initially, the foreign key referencing SESSION_ID in table HIT_COUNTS was unindexed. The following script adds the index online.
Using the above script the index was added to the application while users were 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 `DBMS_REDEFINITION`. 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.

```
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:

```
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 Optimisation (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 requiring default values.

Prior to Oracle Database 11.1 adding a column with a non-nullable default value required an update of the entire table to store the default value after the column is added, causing significant load on the storage system and other overhead. Oracle Database 11.1 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 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 the existing transactions are finished, then briefly lock the table and finish. There is no dedicated ONLINE keyword.

**Using DBMS_REDEFINITION to redefine Tables Online**

Oracle Database provides a mechanism to make table structure modifications without significantly affecting the availability of the table. The mechanism is called online table redefinition. 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 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.

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

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

If no errors or exceptions are thrown 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:

```sql
CREATE TABLE sessions_json (  
    -- this is a UUID|
    session_id          char(32),
    session_data        JSON
);  
```

Note that indexes will be copied automatically as part of the procedure, 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 cope with 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) 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 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 are identical.
- A call to JSON_OBJECT() featuring the relational columns from the source table is mapped to the JSON_DATA column in the interim table.

It is possible to copy the table dependents over as well, this example copies the unique index (not the primary key constraint). Note that statistics must be gathered manually since COPY_STATISTICS is FALSE, a conscious decision given the different structure featured in source and interim 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.

DECLARE
    l_errors PLS_INTEGER;
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;

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

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

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

```sql
BEGIN
    DBMS_REDEFINITION.finish_redef_table(
        uname => user,
        orig_table => 'SESSIONS',
        int_table => 'SESSIONS_JSON',
        dml_lock_timeout => 60,
        continue_after_errors => false
    );
END;
/
```

As soon as the prompt returns you should gather stats on the table, and enable the primary key constraint based on the copied index.

```sql
ALTER TABLE sessions
    ADD CONSTRAINT pk_sessions PRIMARY KEY ( session_id )
    USING INDEX pk_session_hit_counter;
```

```sql
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. When the installation of an 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.
Therefore, existing sessions can continue to use the pre-upgrade application until usage reaches its natural end; and all new sessions can use the post-upgrade application. When there are no longer any sessions using the pre-upgrade application, it 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 make your application more resilient to changes is a win!

**EBR Concepts**

As the name implies, Edition-based redefinition is based around editions. Editions are nonschema objects; as such, they do not have owners. 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 in your application.

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 every type of trigger that you can define on a table (except crossedition triggers, which are temporary, and \texttt{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.

If other users must be able to change data in the tables while you are changing their structure, then you can also use forward crossedition triggers. If the pre- and post-upgrade applications will be in ordinary use at the same time (hot rollover), then you can also use reverse crossedition triggers. Crossedition triggers are not a permanent part of the application—you drop them once all users are using the post-upgrade application. 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 API PL/SQL without incurring library cache locks.

Objects in the library cache such as PL/SQL code are protected from modifications by DDL locks. A PL/SQL stored procedure in use by sessions 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.
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.

**Potential Workflows**

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

1. Editions-enable the appropriate 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:

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. Remember that your application ideally uses an API written in PL/SQL decoupling the frontend from the database backend, therefore breaking the link between their respective release cycles.