

# Understanding the Oracle APEX Application Development Lifecycle

---

Oracle APEX Technical Paper

June 2025, Version 3.2

Copyright © 2025, Oracle and/or its affiliates

Public

## Purpose Statement

This document explains the Oracle APEX Application Development Lifecycle. Examples have been validated using APEX 24.2 and SQLcl 25.1.1 releases.

## Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

## Table of Contents

|                                                                          |           |
|--------------------------------------------------------------------------|-----------|
| <b>Purpose Statement</b>                                                 | <b>2</b>  |
| <b>Disclaimer</b>                                                        | <b>2</b>  |
| <b>Introduction</b>                                                      | <b>6</b>  |
| <b>Overview of APEX Architecture and Application Artifacts</b>           | <b>6</b>  |
| Workspaces Contain Apps Comprised of Components                          | 6         |
| Larger Solutions Can Be a Group of “Mini Apps”                           | 7         |
| Unique Architecture Enables Instant, Scalable Data Access                | 7         |
| Metadata-Driven Execution Engine, Just Like the Database                 | 8         |
| Preserving the History of Your Definitions Is Up to You                  | 8         |
| APEX Development Lifecycle Follows Familiar File-Based Paradigm          | 9         |
| <b>Track Bugs and Features to Plan Milestones</b>                        | <b>9</b>  |
| APEX Team Development is Built Right In                                  | 10        |
| Other Products Offer Kanban Boards for Agile Teams                       | 10        |
| <b>Work on Fixes and Features in Parallel While Minimizing Conflicts</b> | <b>12</b> |
| Understand APEX Multi-User Development                                   | 12        |
| Multi-User Development for Components                                    | 12        |
| Multi-User Development for Pages                                         | 12        |
| Use Page Locking to Prevent Conflicting Changes                          | 13        |
| Lock a Page to Temporarily Take Exclusive Ownership                      | 13        |
| See What Pages are Locked in Your Application                            | 13        |
| View a Locked Page in Page Designer                                      | 14        |
| Unlock a Page to Let Another Team Member to Edit It                      | 14        |
| Use a Working Copy to Isolate Fix or Feature Work                        | 14        |
| Modularize Your App to Further Minimize Conflicts                        | 15        |
| Assemble Larger Solutions from Multiple “Mini Apps”                      | 15        |
| Share Components, Pages, and Packages from a Common App                  | 15        |
| Subscribe, Publish, and Refresh Your Common Components                   | 15        |
| Enable Authentication Across Mini Apps                                   | 16        |
| Consider Separating Schema Objects by Mini App                           | 16        |
| <b>Manage the History of Changes Using Source Control</b>                | <b>16</b> |
| Advantages of Using a Source Code Repository                             | 16        |
| Use Git to Manage Your Application Changes                               | 16        |
| Start By Periodically Committing an Application Export                   | 17        |
| Choose a Hosted Git Repository                                           | 17        |
| Install the Latest Version of SQL Command Line (SQLcl)                   | 17        |
| Save a Named Connection to Simplify Operations                           | 17        |
| Creating a SQLcl Project to Manage APEX Application Lifecycle            | 18        |
| Configuring Project Filters and Settings                                 | 18        |
| Exporting APEX Application and Database Objects for a Milestone          | 19        |
| Staging Branch Changes to Generate Distribution Scripts                  | 20        |

|                                                                         |           |
|-------------------------------------------------------------------------|-----------|
| View the History of Application Changes                                 | 20        |
| <b>Test Business Logic, User Interfaces, and Measure Code Coverage</b>  | <b>22</b> |
| Use utPLSQL to Write and Run Unit Tests for PL/SQL                      | 22        |
| Download and Install the utPLSQL Framework                              | 22        |
| Define Tests Using PL/SQL Packages with Annotation Comments             | 22        |
| Implement Tests in the PL/SQL Package Body                              | 22        |
| Run Tests to Ensure Application Functionality Never Regresses           | 23        |
| Measure Code Coverage to Improve Tests Over Time                        | 23        |
| Exercise End to End User Interface Tests with Cypress                   | 24        |
| Author UI Tests Using English-Like Commands                             | 24        |
| Run and Debug UI Tests Interactively                                    | 25        |
| <b>Release an Installable Application Artifact to Deploy Elsewhere</b>  | <b>26</b> |
| Create a Project Release                                                | 26        |
| Generate a Project Release Artifact                                     | 27        |
| <b>Deploy Artifact of Your App to Test or Production Environment</b>    | <b>27</b> |
| <b>Aspire to Maintain Separate Environments for Dev, Test, and Prod</b> | <b>27</b> |
| Why Separate Environments Are a Best Practice                           | 27        |
| Use Environment Banners to Avoid Inadvertent Mistakes                   | 28        |
| Dev, Test, and Prod in Separate Workspaces in an APEX Instance          | 28        |
| One-Time Project Configuration to Support Test and Prod                 |           |
| Deployment to Different Workspaces                                      | 30        |
| Dev, Test, and Prod in Separate APEX Instances                          | 30        |
| <b>Inspect Code for Performance, Security, or Quality Problems</b>      | <b>30</b> |
| Use APEX Advisor to Run Built-in Audits                                 | 31        |
| Extract SQL and JavaScript Embedded Code for Your App                   | 31        |
| Highlight SQL Injection and Poor SQL Patterns with SQLcl                | 31        |
| Perform Security and Quality Audits with SonarQube and ApexSec          | 31        |
| <b>Understand Team-Centric vs. Feature-Centric Development</b>          | <b>32</b> |
| Overview of Team-Centric Development Approach                           | 32        |
| Overview of Feature-Centric Development Approach                        | 32        |
| Work Privately Using a Git Branch and "Branch Instance" of APEX         | 33        |
| Git Manages Parallel Changes to Files Using Branches                    | 33        |
| Branch Instance of APEX is Dedicated to a Single Feature or Fix         | 34        |
| Walkthrough of Feature-Centric Development                              | 34        |
| Committing and Pushing Changes on a Branch                              | 35        |
| Resolving Merge Conflicts                                               | 36        |
| Initiating Peer Review                                                  | 38        |
| Reviewing the Merge Request                                             | 38        |
| Merging Approved Changes to Main                                        | 39        |
| Updating the Central APEX DEV Instance with the Latest Build            | 39        |
| Preparing a Release to Install in Other Environments                    | 39        |
| <b>Automate the Build, Test, and Deployment Process</b>                 | <b>40</b> |



|                                                                          |           |
|--------------------------------------------------------------------------|-----------|
| Understanding Automation Pipelines                                       | 40        |
| Pipeline to Export, Stage, and Create New Release Artifact               | 41        |
| Pipeline to Create Build, Run Tests, and Deploy to Test Environment      | 42        |
| <b>Conclusion and Recommendations</b>                                    | <b>45</b> |
| Recommendations for Simple Apps and Small Teams                          | 45        |
| Recommendations for Mission-Critical Apps and Larger Teams               | 45        |
| <b>Appendix A: Oracle Cloud Infrastructure Connection Considerations</b> | <b>46</b> |
| SQLcl Requires a Database Connection                                     | 46        |
| Connection to Database in a Private Subnet Requires a Bastion            | 46        |
| <b>Appendix B: Other Ways to Export APEX Apps and Schema Objects</b>     | <b>47</b> |

## Introduction

Oracle APEX is the world's most popular low-code application platform for enterprise apps. It's free to use with any Oracle database or database cloud service. Using a browser-based builder, you and your team create modern data-centric web apps that are reliable, scalable, and secure. End users access them with desktop or mobile browsers or install them like native apps in one click. Every day APEX developers the world over help each other succeed. They produce everything from small departmental apps done by one or two people, to complex, business critical systems engineered by a team of IT professionals.

As users get their work done with your application, they'll often report bugs and suggest improvements to enhance productivity or address new business requirements. Every development team aims to make steady, incremental progress against a prioritized list of these issues and ideas. The development *lifecycle* is the process of periodically selecting a set of bugs and features that will improve the app, dividing the work among teammates, and testing the result before releasing it to end users. Delivering a few change requests at a time to end users by producing a series of high-quality releases on a regular cadence is the goal.

In practice, to successfully *accomplish* this goal you need to understand how to:

- Track details of all bugs and features and which are planned for the current milestone
- Work on fixes or features in parallel while minimizing conflicts
- Manage the history of all changes to application artifacts in a source control repository
- Test application business logic, user interfaces, and measure code coverage
- Create an application archive to deploy later
- Deploy an application archive to a test or production environment, and
- Aspire to use separate APEX environments for development, testing, and production

In addition, larger teams working on more complex applications will benefit from learning how to:

- Inspect code for performance, security, or quality problems
- Isolate work on distinct features using source control branches and “branch instances” of APEX, and
- Automate the build, test, and deployment process to increase productivity.

This paper explains all these aspects of the APEX development lifecycle and highlights their use in popular development lifecycle solutions like GitLab, GitHub, Oracle VB Studio, and Jenkins.

## Overview of APEX Architecture and Application Artifacts

Since the APEX development lifecycle involves managing APEX application artifacts, this section briefly explains the APEX architecture and the artifacts you and your team will create and enhance during APEX development.

### Workspaces Contain Apps Comprised of Components

An APEX workspace contains applications made of components like pages, lists of values, and processes. APEX apps let end users visualize, explore, find, and manage data, so each workspace has an associated database schema. As shown in Figure 1, developers often organize database objects for *Application A* into a schema like *Schema A*. It can contain tables, views, PL/SQL packages, and any other kind of schema object the Oracle Database supports.

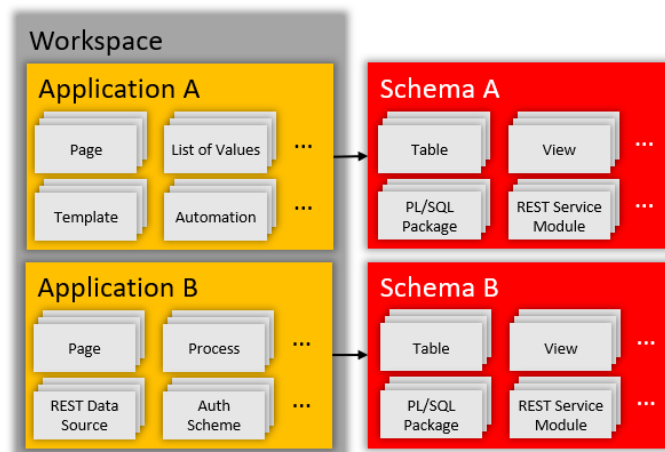


Figure 1: A workspace contains apps comprised of components, and it references schemas

### Larger Solutions Can Be a Group of “Mini Apps”

While smaller solutions don’t require it, you can build larger applications like the conference management system shown in Figure 2 as a group of “mini apps.” One might handle conference speakers, sessions, and schedule. Another manages sponsors and exhibitors. A third one lets attendees register and build an agenda. End users login to the system and seamlessly access pages across mini apps to perform any functions they’re authorized to do. Each mini app subscribes to a set of common components from a “library” app to ensure a consistent user experience. These subscribed components can include menu navigation lists, templates, lists of values, authentication schemes, and more. Many teams create a distinct workspace for each logical application to contain the library app and all the mini apps that comprise it. But if you prefer a single workspace, you can group apps using tags to easily focus your attention in the Builder on a subset of apps that go together.

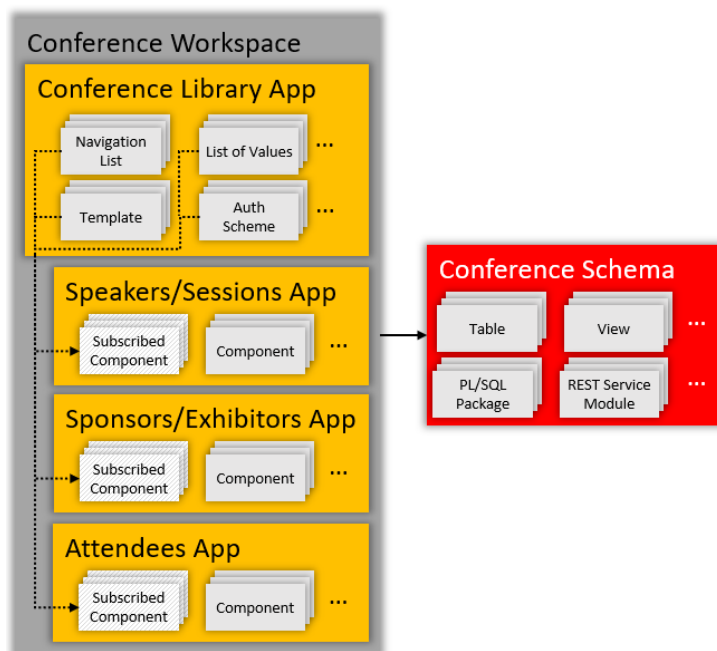


Figure 2: Logical app made of mini apps with component subscriptions

### Unique Architecture Enables Instant, Scalable Data Access

Oracle APEX’s unique architecture enables instant access to local data, flexible access to remote data, and automatic scaling and failover when running on Oracle Autonomous database. As shown in Figure 3, desktop or mobile browsers send HTTPS requests to an Oracle REST Data Services (ORDS) listener that delegates them to the Oracle APEX execution engine running inside the Oracle Database. The APEX engine assembles the response by referencing application metadata for the requested page. In the process, it accesses local data, may retrieve

remote data over HTTPS from REST services, and generates HTML, CSS, and JavaScript to present the data. The ORDS listener relays the response back to the requesting device.

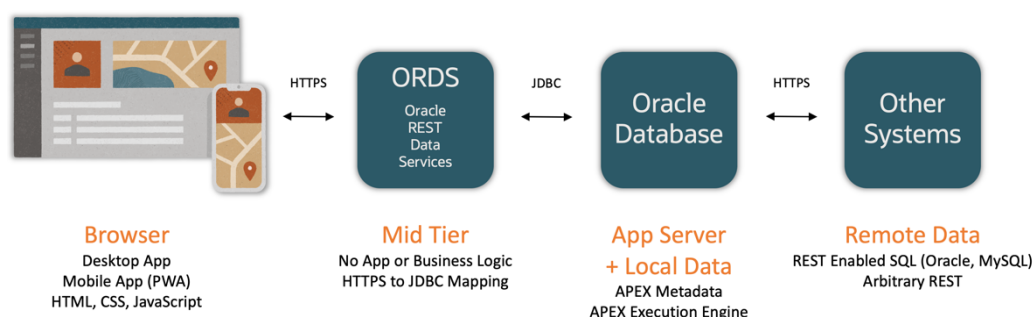


Figure 3: ORDS listener connects desktop & mobile clients to APEX apps executing inside the database, accessing remote REST data as needed

## Metadata-Driven Execution Engine, Just Like the Database

APEX not only runs *inside* the Oracle database, but its metadata-driven execution engine mirrors the way the database works, too. As you define the database schema objects your application requires, the database stores *metadata* about them in its data dictionary and you can query it using dictionary views. For example, to see the name and datatype of a particular table's columns, you can query the USER\_TAB\_COLUMNS view. The APEX Builder does the same thing while you define your APEX application components. It saves information about them into the *APEX* data dictionary and lets you query it using APEX dictionary views. For example, to find the name and type of items on a particular page in your application, you can query the APEX\_APPLICATION\_PAGE\_ITEMS view.

Once you've defined the columns in a table, the database engine references that metadata when performing operations on the table. You express *what* you want to do (e.g. SELECT, INSERT, UPDATE, etc.) while the engine handles *how* to get the job done. As Oracle iteratively improves release after release, operations on your existing tables get faster with no changes required to the *definitions*. The same holds for all APEX components, too. Consider a chart component, for example. Once you've defined *what* data you want to see and the style of its presentation, the APEX engine handles *how* to render that chart. Over time, APEX can continually improve how the chart is implemented to embrace the latest techniques and technologies with no changes required to your chart component's *definition*. For example, the bars in your trusty bar chart might begin to animate up from the x-axis in the latest APEX release, with no work required on your part to benefit from this improvement.

## Preserving the History of Your Definitions Is Up to You

While you will change your database objects and APEX application components over time, the respective data dictionaries reflect only the *current* definitions of schema objects and APEX components in use right now. Preserving the history of your team's changes to these definitions requires a conscious effort on your part. This entails archiving a point-in-time snapshot of simple text files representing the definitions each time the project reaches a meaningful milestone. In addition to the SQL scripts, you may use to create your database schema objects, every APEX application and component has an equivalent text file representation, too. You can easily retrieve the text files representing your application at any time. As shown in Figure 4, exporting the current state of your application produces a zip archive containing SQL scripts representing the application and each component it contains. The export can also include optional YAML and code extract files that simplify comparing what's changed in the latest version and performing scans on any SQL, PL/SQL, or JavaScript code your application components include. The text files comprising your database object definitions and those of your APEX components are known as your *application artifacts*.

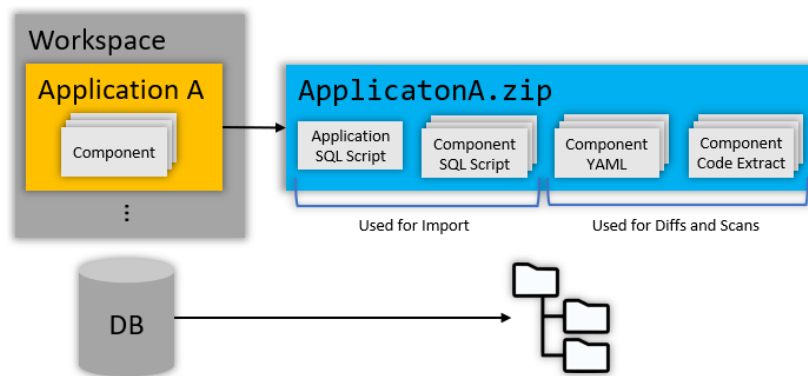


Figure 4: Exporting an APEX application to produce archive of component files

## APEX Development Lifecycle Follows Familiar File-Based Paradigm

Methodically storing the history of your application artifacts over time lets you install any version of your application at any time on any target system for any purpose. Installing database schema object definitions involves running the relevant SQL scripts in an opportune order to create tables, views, PL/SQL packages, and more. Similarly, you install a particular version of your APEX application by importing an archive of application and component scripts as shown in Figure 5.

The history of application artifact snapshots you’ve taken over time represents the “source of truth” for your application. In your production environment, the version installed in the database is the current one end users are using. In your testing environment, it might be the most recent weekly snapshot of your application’s next release. And in your development environment, the version installed is undergoing active development. Read on for an explanation of the best practices you and your team need to know to effectively deliver improved versions of your APEX application over time, with high quality, a few changes at a time.

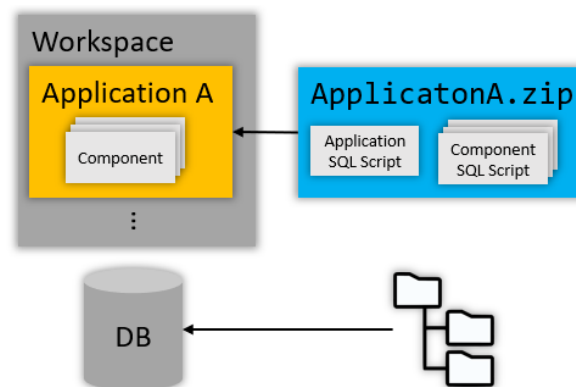


Figure 5: Importing APEX application from the file system into the database

## Track Bugs and Features to Plan Milestones

It’s important to track any defects found in your app as well as any enhancements to improve it. An issue tracking system lets you document the steps to reproduce a bug and provide background on the goals of a new feature, attaching images and other documents to supplement teammates’ comments as needed. A flexible system helps you classify, prioritize, and assign issues to developers on the team to tackle for an upcoming milestone. Above all, having a system in place ensures that no good idea or annoying issue gets forgotten, even if the team can’t address it immediately. Oracle APEX features a built-in issue tracker, while Oracle VB Studio provides more complete support for agile teams. GitHub Issues, GitLab Issues, and Jira provide popular alternatives.

## APEX Team Development is Built Right In

APEX comes with built-in *Team Development* that provides the basic functionality your team needs to manage issues, label them in a flexible way, group them into milestones, and assign them to teammates. Figure 6 shows the Team Development home page with issues tagged as bugs or feature requests. It also shows the associated milestone and the team member who owns each issue.

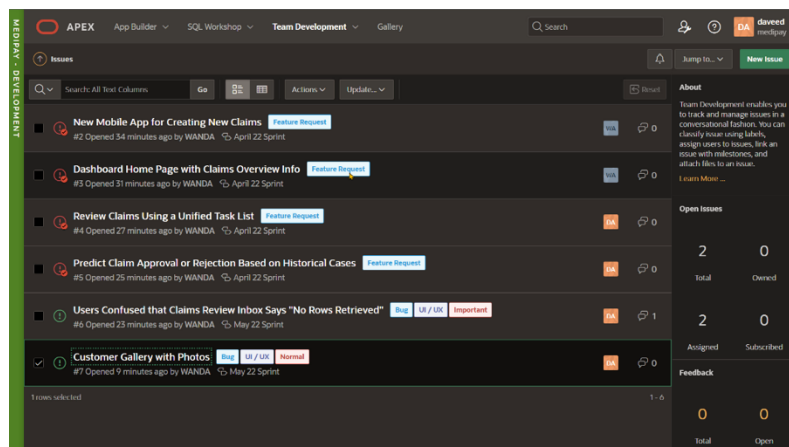


Figure 6: APEX Team Development provides built-in issue-tracking with labels and milestones

All the issue tracking systems mentioned here let team members comment over time on issues, including basic formatted text, attached documents, or images to enrich the discussion. Figure 7 shows what this looks like for an APEX Team Development issue.

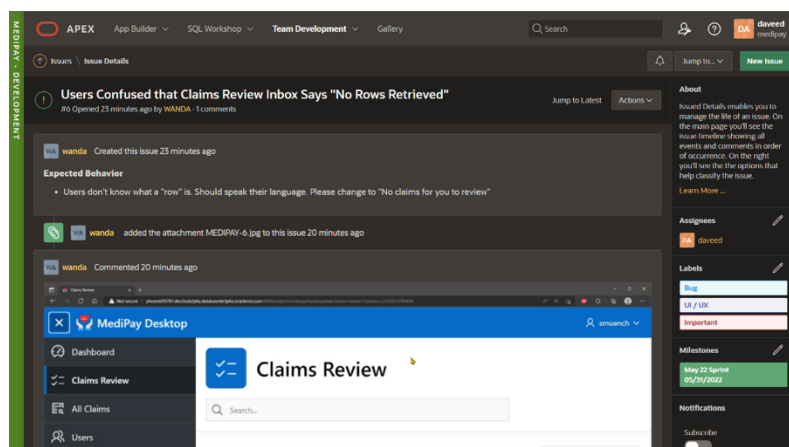


Figure 7: Issue details page with comment history including images, labels, milestone, and assignee

Developers can easily filter the list of issues assigned to them, optionally saving their filtered lists for quick reuse. A similar facility lets teammates see issues assigned to a particular milestone and who is working on each one.

## Other Products Offer Kanban Boards for Agile Teams

In addition to supporting all the issue-tracking table stakes features, Oracle VB Studio's issue tracker lets teams follow an agile scrum methodology by visualizing their backlog and dragging issues into a milestone called a "sprint". As shown in Figure 8, team members use a Kanban board for each sprint with vertical "swim lanes" showing issues to work on, ones currently in progress, and those already completed.

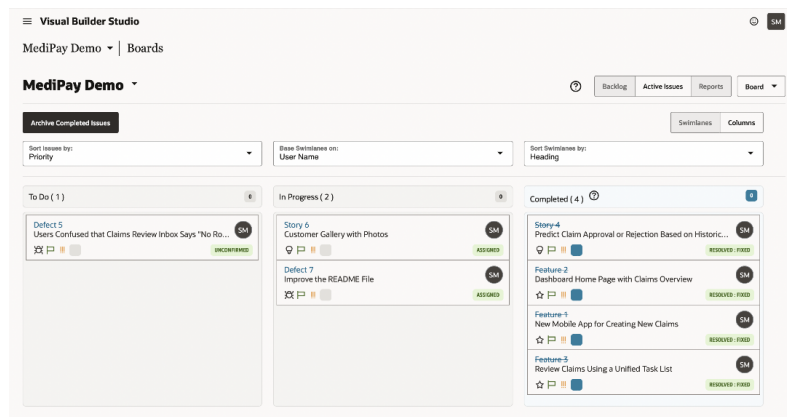


Figure 8: Oracle VB Studio's issue tracker supports Kanban boards

Figure 9 is Atlassian's *Jira*, another popular issue tracker with agile Kanban boards and backlog management.

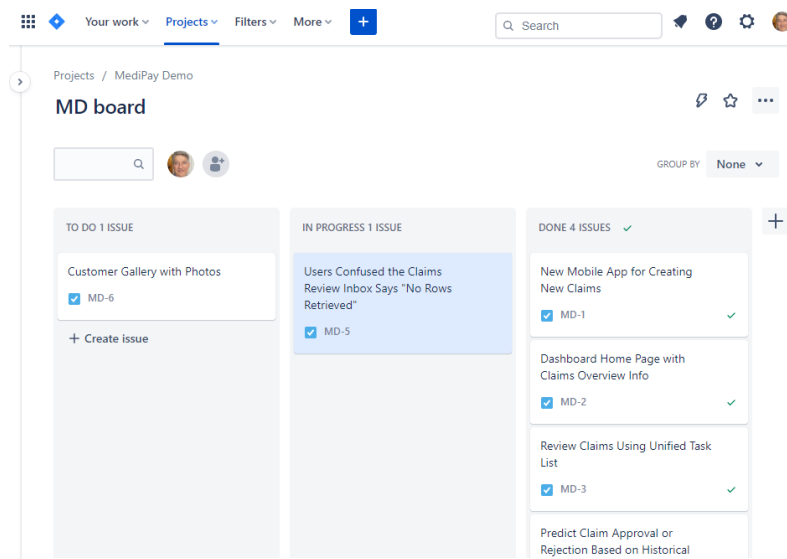


Figure 9: Jira showing Kanban board of issues

As a final example, *GitHub Issues* in Figure 10 supplements the Kanban board view with an editable table view of the sprint's issues.

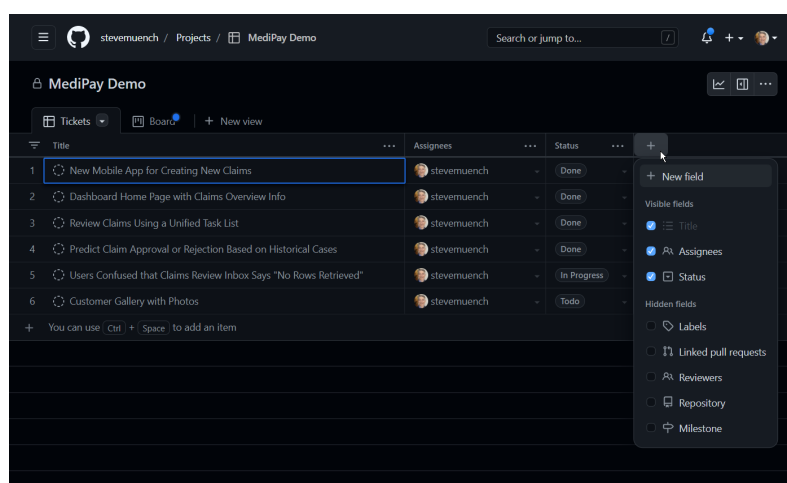


Figure 10: GitHub Issues offers a tabular view of project issues in addition to a Kanban board

Once you've chosen an issue tracker, entered your bugs and features into the system, defined an upcoming milestone, and assigned issues to teammates, you and your team are ready to start making progress at implementing the set of issues in the current "sprint".



## Work on Fixes and Features in Parallel While Minimizing Conflicts

Oracle APEX lets teams of developers work in parallel on a shared development instance and provides page locking, lost update protection, and working copies to help them avoid stepping on each other's toes. For example, a developer can create a working copy of an app to fix a bug or build a feature in isolation until they are ready to merge it back into the main application. It is important to understand how APEX multi-user development works, how to use page locking and working copies, and how you can modularize your app to minimize conflicts.

### Understand APEX Multi-User Development

Oracle APEX is a multi-user development environment with the current version of your application's metadata stored in the database. APEX developers use their web browser to create and modify application pages using the Page Designer. They use wizards and edit pages to work on all other kinds of components. All aspects of the APEX design-time experience let multiple team members create and edit application components at the same time. When potential conflicts arise between different developers' changes, APEX errs on the side of caution so it's good to know how APEX will behave when you and a teammate might "bump" into each other in your daily work.

### Multi-User Development for Components

While you and your team use APEX to implement your fixes or features, consider two situations that might arise due to unlucky timing of two colleagues' actions. Imagine that both you and a colleague perform the following change at around the same time:

- Both create a List of Values named CUSTOMERS\_LOV
- Both edit the same RecentPayments REST Data Source

In both scenarios, one of the developer's changes will be saved first, and the other developer will receive an error message saying for example:

- *"CUSTOMERS\_LOV: ORA-00001: Unique constraint violated"*
- *"Current version of data in database has changed since user initiated update process"*

APEX raises these errors to prevent one developer's changes from silently overwriting another's change that was made at nearly the same time. The developer receiving the error will need to try again, perhaps realizing in the process that the work was already done by a colleague.

### Multi-User Development for Pages

The Page Designer also prevents developers from overwriting each other's work, but it enforces its protection at a more granular level than the component wizards and edit pages. The Page Designer tracks changes to every property of every page component and saves only what has changed. If two developers editing the same page at the same time each change *different* page components, or change different *properties* of the same page component, then both of their respective sets of changes will save correctly. The result will be the union of those modifications. While teammates are making changes to the same page, they do **not** see each other's unsaved changes "live". But when a developer saves her own changes to the page, the Page Designer will refresh to reflect any changes colleagues might have saved in the meantime.

However, if two colleagues working in Page Designer on the same page happen to both change the same property of the same page component, then the one who saves first will succeed while the second one receives an error *"Saving changes failed! See messages for details."*

As shown in Figure 11, when this occurs the conflicting properties are identified with a circular red "x" icon in the Property Editor and the *Messages* window indicates which property values have conflicted with recently saved changes by a colleague. The developer encountering this situation won't be able to save the page, but her original values are preserved in case she wants to copy them to an alternative location before abandoning her current edits and attempting to make the changes to the current page again.



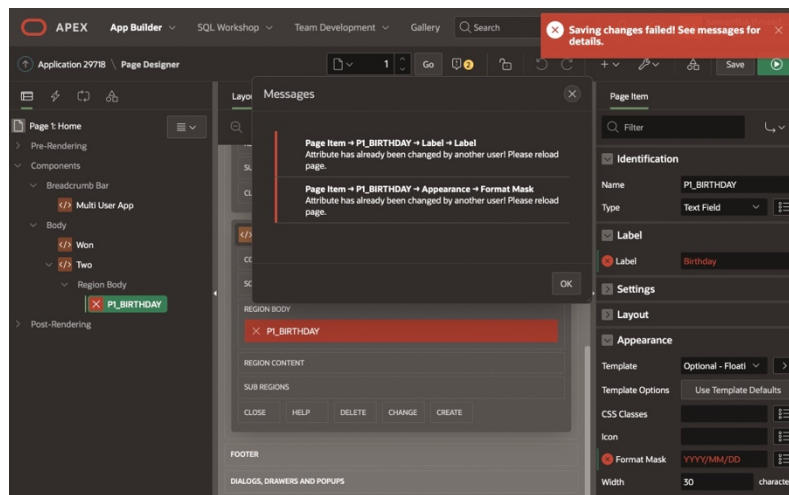


Figure 11: Page Designer showing multiple conflicting property changes

## Use Page Locking to Prevent Conflicting Changes

After doing significant work on a page, imagine the frustration of clicking the **Save** button only to encounter a “*Saving changes failed!*” message. While you are working on enhancing the page’s functionality, suppose a colleague opens the same page and saves a quick edit to one of the same page component properties you are busy changing. This would inadvertently cause the “*Saving changes failed!*” message when you later save the page. In this scenario, you’d sadly be forced to abandon your changes to that page and to do them all again. While the multi-user edit features described above offer useful protection, it’s best-practice to avoid making conflicting edits in the first place.

Communicating with teammates using a wiki page, email, or Slack is one way to let others know what you’re working on so you all can avoid unintentionally disrupting each other’s work. However, the page locking feature is an even *more* effective way to develop defensively so your saves always go smoothly.

## Lock a Page to Temporarily Take Exclusive Ownership

Oracle APEX page locking lets one developer claim exclusive edit rights to a page while they are working on some important changes. The duration of the lock is flexible. It might be for mere minutes, or could be locked for weeks to accommodate extensive enhancements to an existing page’s functionality. Other team members see the page is locked and can still view the page in the Page Designer. However, they can’t edit it again until the page is unlocked. As shown in Figure 12, after clicking the lock icon to lock the page you can enter a helpful comment to explain why you are doing it.

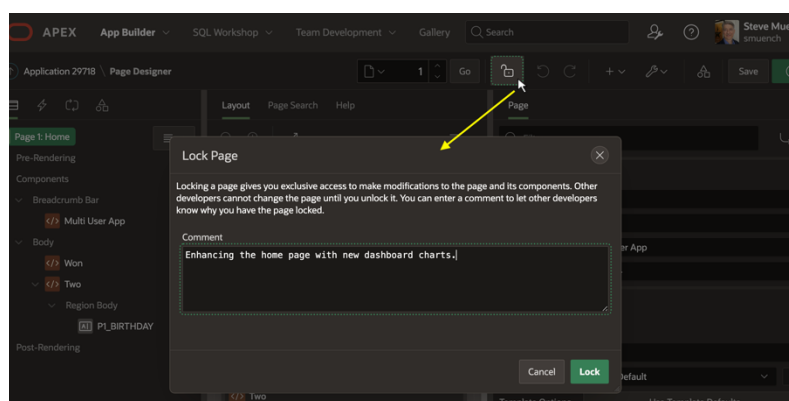


Figure 12: Providing a useful comment when locking a page

## See What Pages are Locked in Your Application

On the application page list, in the *View Report* mode team members see a tabular list containing a *Lock* column that shows an appropriate icon next to any locked pages as shown in Figure 13.

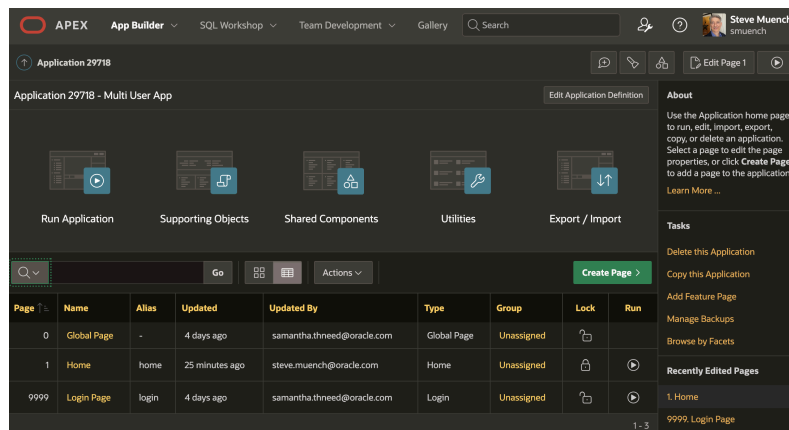


Figure 13: Application page list shows locked pages with an icon

## View a Locked Page in Page Designer

When a teammate *visits* the page you locked in Page Designer, the editor is view-only and the locked icon confirms it. Hovering her mouse over it shows which colleague is currently working on the page. As shown in Figure 14, she can also *click* the locked icon to see the comment explaining what her teammate is working on.

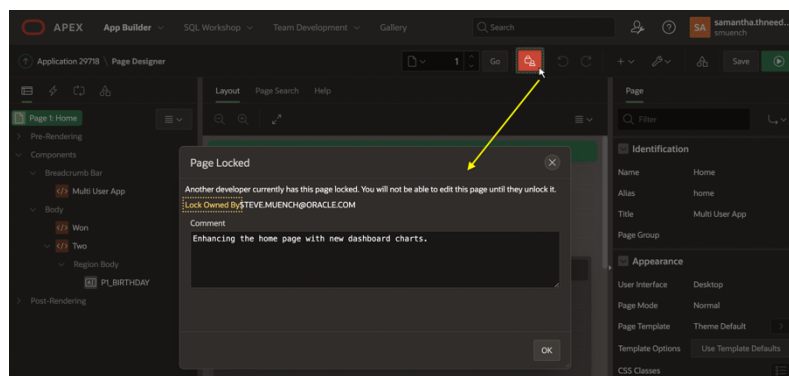


Figure 14: Colleagues see you locked the page and can view your lock comment

## Unlock a Page to Let Another Team Member to Edit It

Once the developer who locked the page is done working on it, she can click the locked icon again in the Page Designer to unlock it. When performing this step, she can adjust her original comment if necessary. After that, another team member can lock the page and work on a new issue involving that page.

## Use a Working Copy to Isolate Fix or Feature Work

At any time, any developer on the team can create a working copy of an app to change it in small, large, or even experimental ways. Using their working copy of the main application, a developer works in isolation of their teammates on a bug fix, new feature, or to explore some changes to flesh out an idea. While working on making the changes in their working copy, a developer can compare it with the main application or with another developer's working copy to study the differences. If teammates merge working copy changes into the main application in the meantime, a developer can refresh their working copy to pick up those changes. Once their work is complete, the developer can decide to merge their changes back into the main application, or possibly just delete the working copy if the changes were experimental in nature.

Using working copies, application changes are merged and refreshed at page or component-level granularity. So lock a page in the main app before changing it in a working copy. This helps teammates know to avoid making *simultaneous* changes to the same page in a different working copy.

## Modularize Your App to Further Minimize Conflicts

In addition to using page locking and working copies, you also can modularize a larger-size application into multiple, smaller “mini apps” to further minimize conflicts between developers.

## Assemble Larger Solutions from Multiple “Mini Apps”

While smaller ones don’t require it, creating your larger solutions as a set of smaller applications lets you keep each “mini app” to a reasonable number of pages for a small team to own. When the team working on an application is small, each team member knows what the others are doing so it’s easier to avoid stepping on each other’s toes. Changes to application logic in PL/SQL packages specific to a given mini app won’t risk conflicting with changes made to the application logic of other mini apps.

## Share Components, Pages, and Packages from a Common App

Of course, some pages, components, and application logic will be useful to all your mini apps so you can organize those shared elements into a “common” app. Mini apps can subscribe to components from this library app, branch to its pages, and invoke PL/SQL APIs from its packages. By clearly identifying common elements from the outset, developers know changes to common components, pages, or code may affect everyone on the team. By organizing the reusable, common components into a component group in the library app, your other mini apps can subscribe to an *entire group* of components in a single operation.

## Subscribe, Publish, and Refresh Your Common Components

In a mini app, to subscribe to a component like a list from the common app, use the *Copy* button from the shared component’s list page and indicate that you want to copy the list from another application. As shown in Figure 15, select the common application in the *Application* list, pick the list to copy, and ensure the *Subscribe* switch is on. Since a *Navigation Menu* is just a list used for this particular purpose, this is the procedure you’d follow in each mini app to have them all share a common navigation menu list, for example.

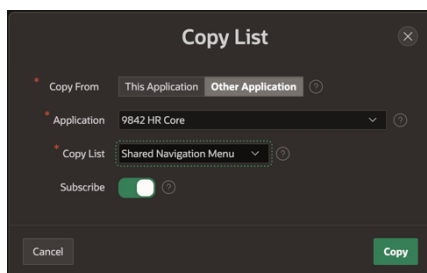


Figure 15: Subscribing to a list from a common app

For other kinds of shared components like List of Values, you subscribe as part of the create wizard. After indicating you want to create a copy of a List of Values from another application and choosing the common app, Figure 16 shows how you can set the *Copy and Subscribe* action for the LOVs you want to subscribe to.

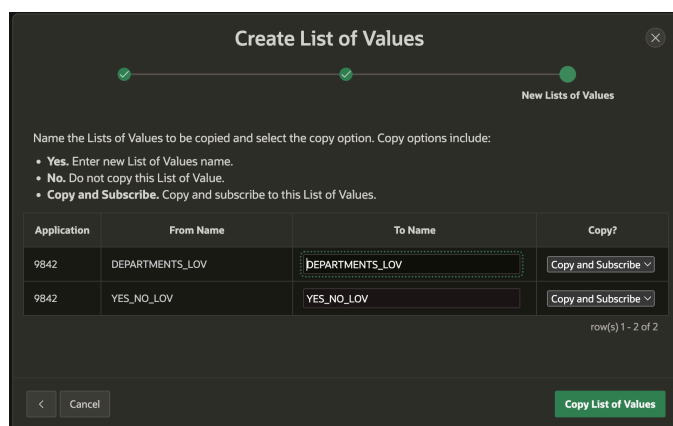


Figure 16: Subscribing to one or more List of Values from the common app

When someone on your team makes changes to the subscribed component in the common app, you decide when to update the subscribing applications. Clicking the *Publish* button on the subscribed component updates its

definition in all the subscribing applications. Alternatively, each subscribing component has a *Refresh* button to click to pull the updated component definition into that specific mini app. The same facilities are available to subscribe to, publish, and refresh component groups for simpler maintenance where useful.

## Enable Authentication Across Mini Apps

By default, each APEX application authenticates its own distinct session. However, if you have modularized your solution into separate mini apps, you'll want end users to navigate seamlessly to any pages in the single *logical* application they use to get their job done. To achieve this, as shown in Figure 17 set the *Session Sharing* type to *Workspace Sharing* in the authentication scheme that your mini apps and common app are using. After configuring this, end users login once and can visit any page across your modularized application.

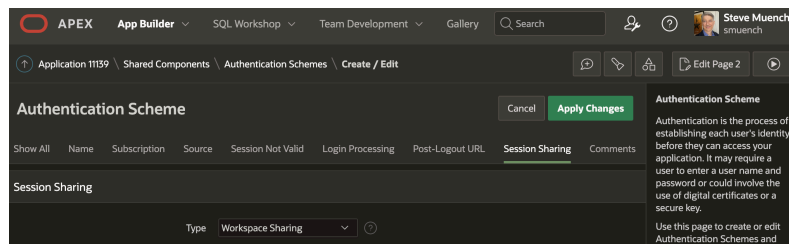


Figure 17: Setting the Session Sharing type on your authentication scheme

## Consider Separating Schema Objects by Mini App

While not required, some teams prefer to have each mini app use its own separate database schema specific to that module. They use appropriate grants and synonyms to let one mini app selectively expose its tables, views, and PL/SQL packages to other modules as necessary. This approach gives mini app developers more control over which schema objects they can consider “private” to their own module, and which they have made “public” to other teams’ mini apps by granting appropriate privileges. A team adopting this technique can confidently make changes to their module-private schema objects, knowing they won’t inadvertently impact other modules.

## Manage the History of Changes Using Source Control

Nobody likes to lose work, so most developers and teams will create periodic backups of their work in progress. In fact, Oracle APEX *also* systematically backs up your applications, too. However, tracking the history of changes to application artifacts over time in a repository offers many additional advantages. As you’ll see in this section, it’s easy to do by combining a source control management (SCM) system like Git with Oracle’s SQLcl tool.

### Advantages of Using a Source Code Repository

A source code repository is like a time machine for your project. An SCM system automatically maintains additional “bookkeeping” information about the project files so at any time, for any change to any application file over the life of your project, you can easily answer questions like: *Who made the change? What got added or modified? When was it made?* And perhaps most importantly, *Why was it made?* Your application’s source code repository provides the historical context about who on the team fixed what bug or implemented which new feature and knows the application artifacts that were added or changed in the process.

The repository becomes the source of truth for your project, letting you tag a particular set of artifacts that got released to end users with a meaningful name like “Release 1.0.3” each time the team hits a major milestone. At any time later, you can use a release tag name to go back in time to retrieve exactly the right set of application artifact file versions to confidently apply an incremental bug fix to a previous version of the application. Seeing the change history grouped by logical feature or fix also simplifies determining what changes are needed to rollback a specific feature or to introduce that feature into an earlier version of your application.

### Use Git to Manage Your Application Changes

[Git](#) is a free source control management system that is the de-facto standard for teams doing development all over the world. Its open-source nature and popularity among other open-source development projects has won it well-earned trust since it debuted in 2005. The permanent copy of the team repository resides on a server. It can

be “cloned” onto another machine for use as a local work area to add, change, or delete project artifacts without immediately affecting the permanent team repository. When pending local changes to project artifacts in the work area are ready to be made permanent, like a database transaction, they get “committed” along with a log message explaining the bug fix or feature was implemented. As a final step, the committed changes get “pushed” up to the permanent team server. The local Git working copy can be deleted at this point, or it can be kept around and refreshed to include any later changes made to the permanent team repository by “pulling” those changes on demand.

## Start By Periodically Committing an Application Export

A survey of APEX developers in the summer of 2022 revealed that two thirds are using some form of source control management system. However, this means that a third of APEX applications are missing a detailed change tracking history. This section shows APEX developers not currently using Git how to get started by periodically committing an application export to a hosted team repository.

## Choose a Hosted Git Repository

While it's *possible* to install the Git server software yourself on-premises, an even easier way to get started is to create a private team repository on one of the Git hosting services available. Both [GitHub](#) and [GitLab](#) are popular Git hosting services whose free tier offerings appeal to smaller teams. Larger teams can upgrade to paid options. APEX projects using Oracle Cloud may appreciate evaluating its native options for a hosted Git repository: [Oracle VB Studio](#) and [OCI Dev Ops](#). A single provider for hosted app development lifecycle services may meet your needs best, and both OCI-based options are free to OCI customers, who pay only for storage and compute for build jobs.

With a Git repository created, the job is half done. Each time the team reaches a meaningful milestone, just nominate a team member to export the application definition and database object definitions from the development database to a local Git work area, and commit the changes to the permanent team repository in Git. The following sections describe the commands to accomplish these steps. They work the same way no matter where you've chosen to host the permanent team repository.

## Install the Latest Version of SQL Command Line (SQLcl)

Using the Oracle [SQL Command Line](#) utility (SQLcl) you can export your APEX application as well as database schema object changes to text files on the file system in a Git work area to commit the changes to a team repository. On all platforms, the simplest way to install SQLcl is to:

- ensure Java 11, 17, or 21 is installed first, and then
- download and unzip the latest version from:  
[https://download.oracle.com/otn\\_software/java/sqldeveloper/sqlcl-latest.zip](https://download.oracle.com/otn_software/java/sqldeveloper/sqlcl-latest.zip)

The SQLcl utility is the **sql** program in the `./sqlcl/bin` subdirectory, which you can add to the system path to invoke **sql** from anywhere. Alternatively, on Mac if you use the brew package manager, you can **brew install sqlcl**, or on Oracle Linux use **yum sqlcl** (OL7) or **dnf sqlcl** (OL8). The examples in this paper have been validated using SQLcl version 25.1.1 and APEX 24.2.

## Save a Named Connection to Simplify Operations

After running SQLcl with the connection credentials of the database schema related to your APEX application, use the connect command to securely save a named connection named dev for your development environment:

```
| $ sql username/password@host:port/service
| SQL> connect -save dev -savepwd
```

Once you've saved a named connection like dev, you can connect using the name with the following syntax from the command line:

```
| $ sql -name dev
```

Or from within SQLcl, you can connect to a named connection (dev) using:

```
| SQL> connect -name dev
```

## Creating a SQLcl Project to Manage APEX Application Lifecycle

The SQLcl **project** command simplifies managing the APEX application lifecycle. It automates the best practices Oracle teams use to source control one or more APEX apps related to a project along with the related database objects on which these applications depend.

After creating a Git repository *yourproject* to store your project artifacts and cloning it to your local machine, change to the *yourproject* directory and initialize SQLcl project management using the `project init` command:

```
$ cd yourproject
yourproject $ sql -name dev
SQL> project init -name yourproject -schemas yourschema
```

This creates project-related files in the *yourproject* directory in the `.dbtools`, `src`, and `dist` subdirectories.

```
yourproject $ tree
.
├── .dbtools
│   ├── filters
│   │   └── project.filters
│   ├── project.config.json
│   └── project.sqlformat.xml
├── .gitignore
├── README.md
├── dist
│   └── install.sql
└── src
    ├── database
    └── yourschema
```

## Configuring Project Filters and Settings

After enabling project tracking for your APEX application, by default SQLcl assumes the project includes:

- **all** APEX applications in your workspace,
- **all** database objects in the schemas you mentioned, and
- **all** ORDS modules.

To *fine-tune* which application ids and database schema objects participate in the project tracking, add the following additional lines to the end of your project's `.dbtools/filters/project.filters` file:

```
-- Export only schema objects with the prefix 'SCH_%'
object_name like 'SCH\_%' escape '\',
-- Export only APEX application 146
application_id in (146),
```

Notice the lines have a SQL WHERE clause predicate syntax, with a trailing comma, and that you can use the PL/SQL double-dash for adding a comment line. The example above includes only application id 146 as well as all database objects whose name starts with the prefix `SCH_` as part of this project. If *your* project consists of multiple “mini apps”, then list their application IDs in the comma-separated `IN` clause for the `application_id` predicate.

SQLcl 25.1.1 considers all ORDS modules as part of the project with no module name filtering. To disable the export of all ORDS modules, perhaps to manually add selected ones by hand as custom SQL files, add the following *additional* lines to the end of your project's `.dbtools/filters/project.filters` file:

```
-- ORDS export is full schema only. Disable ORDS schema export
export_type not in ('ORDS_SCHEMA')
```



By default, SQLcl considers the database objects' *schema* name an important detail to include in the export, and does *not* export your APEX applications' supporting objects. To change this behavior, edit the following details in your project's `.dbtools/project.config.json` file. To *exclude* the database schema name, set the `setTransform > emitSchema` property to `false`.

```
⋮
"setTransform": {
  ⋮
  "emitSchema": false,
  ⋮
}
```

Normally, you want SQLcl to export the database object definitions you've identified with your filters configured in the previous section. However, if you use APEX's *Supporting Objects* to manage your application's database object installation and upgrade scripts, then you will need to ensure `project.filters` exports only APEX applications and also adjust a setting here. To *include* APEX supporting objects, set the `apex > expSupportingObjects` to `Y`.

```
⋮
"apex": {
  ⋮
  "expSupportingObjects": "Y",
  ⋮
}
```

After making the changes to the project tracking filters and settings, add and commit the project files to the main branch of your Git repository.

```
yourproject $ git add --all
yourproject $ git commit -m "initialize project tracking"
```

## Exporting APEX Application and Database Objects for a Milestone

From the *yourproject* directory, in the context of a Git branch representing the current milestone's work, use SQLcl's **project export** command to export the APEX applications and database objects in the project.

```
$ cd yourproject
yourproject $ git checkout -b "milestone-1"
Switched to a new branch 'milestone-1'
yourproject $ sql -name dev
SQL> project export
```

SQLcl reports progress to the console, so you'll see something like the following:

```
The current connection host:port/service YOURSCHEMA will be used for all operations
*** TABLES ***
*** PACKAGES ***
⋮
*** APEX_APPLICATIONS ***
Exporting Workspace YOURWORKSPACE - application 146:School Course Catalog Management
-----
APEX_APPLICATION          1
⋮
TABLE                      4
-----
Exported 16 objects
Elapsed 11 sec
```

After checking the exported files, add and commit them to the current branch:

```
yourproject $ git add --all
yourproject $ git commit -m "milestone-1: team progress for week"
```

### Staging Branch Changes to Generate Distribution Scripts

SQLcl automatically manages a sequence of Liquibase scripts used to distribute your project's application and schema object changes to other environments. Liquibase executes a sequence of scripts against a target database and tracks which ones it applies in each target environment. SQLcl `project stage` command produces a sequence of appropriate distribution scripts to apply incremental database object changes. At project deployment time, it relies on Liquibase to run the scripts in sequence that have not yet been applied to the current target database. This strategy lets SQLcl deploy appropriate database changes required to bring any target database into sync with the latest state of the database objects in your project's `./src` directory.

Use the **project stage** command to compare the committed contents of the current milestone branch with the main branch to generate the necessary scripts.

```
$ cd yourproject
yourproject $ git status
On branch 'milestone-1'
yourproject $ sql -name dev
SQL> project stage
```

SQLcl reports the progress to the console, so you'll see something like:

```
Stage is Comparing:
Old Branch refs/heads/main
New Branch refs/heads/milestone-1
:
Stage processing completed, please review and commit your changes to repository
```

After checking the staged files in the `./dist` subdirectory, add and commit them to the current branch:

```
yourproject $ git add --all
yourproject $ git commit -m "milestone-1: add distribution scripts"
```

Once this milestone's changes are tested to your satisfaction and have passed any formal review process your team may have in place, the milestone branch can be merged into the main branch in Git.

### View the History of Application Changes

With a Git repository as the source of truth for your application, it's easy to get a bird's eye view of all the changes to the project. Many tools support Git, and one particularly popular one is the free Visual Studio Code editor. By installing the free [Git Graph](#) extension, as shown in Figure 18, you can quickly visualize the history of all changes made to your application. It's organized chronologically by commit transactions, and the helpful comment provided at the time each commit was made lets you see what bugs and features were added over time. Clicking on a particular commit expands it to show the files changed as part of that "transaction".



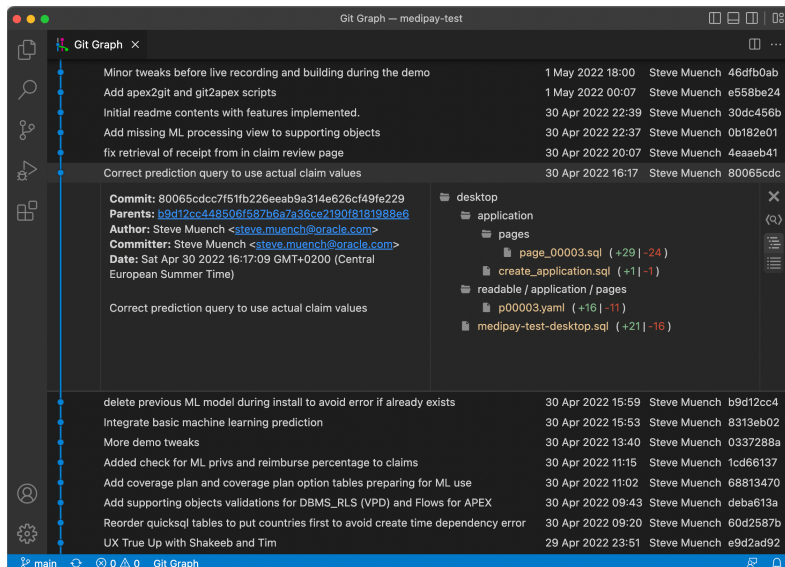


Figure 18: Git Graph in VS Code showing Git history log for an APEX application

Clicking on any filename for any commit, you can easily see what the developer changed in that file. Figure 19 shows the Visual Studio Code file comparison tab that opens when clicking on the `p00003.yaml` file in the above Git Graph screenshot for the expanded commit with message “Correct prediction query to use actual claim values”. It uses color to highlight the changes made to the SQL statement related to a classic report in Page 3 to implement the display of a machine learning prediction based on actual insurance claim data instead of the hard-coded values that were previously used. Even if Page 3 may have been changed many times since the feature related to this commit was implemented, Git and Visual Studio Code cooperate to show the changes based on the previous version of the Page 3 metadata as it existed in the repository at the time the change was made.

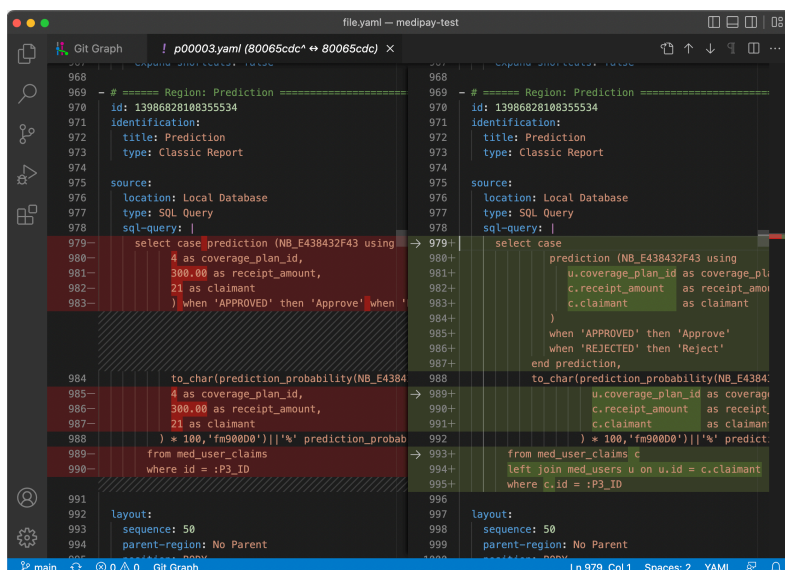


Figure 19: Side-by-side diff of the Page 3 readable YAML file changed in a particular commit

## Test Business Logic, User Interfaces, and Measure Code Coverage

While ad hoc testing is invaluable, nothing gives a development team more confidence to innovate than a broad set of unit tests they can run regularly to ensure all key application functionality continues to work as advertised. Best practice dictates that most application business logic should be organized into PL/SQL packages, and the tests should exercise all aspects of these packages. Code coverage metrics inform future testing efforts so teams can steadily raise the percent of all code paths their unit tests exercise over time. Where possible, automated user interface testing can shoulder some of the burden to let your team's manual testing focus on more complicated scenarios.

### Use utPLSQL to Write and Run Unit Tests for PL/SQL

The open-source [utPLSQL](#) framework is the most popular way to unit test PL/SQL code. Tests are easy to write, easy to run, and easy to automate: a compelling combination that inspires team members to create lots of tests.

### Download and Install the utPLSQL Framework

After downloading the `utPLSQL.zip` file for the latest release from the [utPLSQL GitHub Releases Page](#), unzip it and run the `install_headless.sql` script as a DBA user for the most typical installation. This ensures the utPLSQL test runner lives in its own schema (UT3), is usable by all other database schemas, and keeps a clean separation between your own application's database schema objects and utPLSQL own objects. See the Install Guide in the distribution for an explanation of other ways it can be installed.

### Define Tests Using PL/SQL Packages with Annotation Comments

A test suite is a group of related tests. You define one by creating a PL/SQL package specification that identifies which procedures are test cases using special comments as annotations. They include a percent sign right after the double-hyphen, immediately followed by an annotation name.

For example, this is the annotation to assign a descriptive name to the test suite package. Note the blank line following it. It is important!

```
--suite(APEX Apps Person Tests)
```

The annotation that precedes a procedure to identify it as a test case is:

```
--%test
```

Other frequently used annotations include:

- **beforeall** for suite setup and **afterall** for suite cleanup
- **beforeeach** for test setup and **aftereach** for test cleanup
- **rollback** to define suite or test rollback behavior

The PL/SQL package specification below defines a test suite with two tests:

```
create or replace package ut_person_pkg is
  --%suite(APEX Apps Person Tests)

  --%test
  procedure try_validating_date_not_in_the_future;
  --%test
  procedure try_validating_us_phone_number;
end;
```

### Implement Tests in the PL/SQL Package Body

You *implement* a test suite by writing the procedure bodies defined in the PL/SQL package spec. The package body below shows the two test procedures defined by the `--%test` annotations above. The test case procedures use utPLSQL's `ut.expect()` function and chained functions `to_be_true()`, `to_be_false()`, and `to_be_null()` to assert the expected values different expressions should have. These are just three examples of many such `to_be_*` functions available. The test succeeds if all its assertions are true and fails if any is false.

```

create or replace package body ut_person_pkg is
  procedure try_validating_date_not_in_the_future is
    l_tomorrow date := sysdate + 1;
  begin
    ut.expect(person_pkg.validate_date_not_in_the_future(l_tomorrow)).to_be_false();
  end;
  procedure try_validating_us_phone_number is
  begin
    ut.expect(person_pkg.validate_us_phone_number('(123) 456-7899')).to_be_true();
    ut.expect(person_pkg.validate_us_phone_number('1234567890')).to_be_false();
    ut.expect(person_pkg.validate_us_phone_number('')).to_be_null();
    ut.expect(person_pkg.validate_us_phone_number(null)).to_be_null();
  end;
end;

```

## Run Tests to Ensure Application Functionality Never Regresses

You can run all the utPLSQL test suites defined in the current database schema using the `ut.run()` procedure. You can use it directly in the APEX SQL Commands window like this:

```
begin ut.run(); end;
```

Alternatively, from the command line you can use SQLcl to run them like this:

```

$ sql -name dev
SQL> set serveroutput on
SQL> exec ut.run

```

The results print to the console like this:

```

APEX Apps Person Tests
  try_validating_date_not_in_the_future [.002 sec]
  try_validating_us_phone_number [.002 sec]

Finished in .018424 seconds
2 tests, 0 failed, 0 errored, 0 disabled, 0 warning(s)

```

The utPLSQL framework also offers a convenient `utplsql` [command line utility](#) to run tests. It also comes with [optional formatters](#) to produce your test results in a format that popular build automation solutions like Jenkins, Oracle VB Studio, and others can turn into HTML pages. The example below runs the unit tests and spools XML-formatted test results into a file `results.xml` in the current directory. Notice the optional argument `ut_junit_reporter()` passed to the `ut.run()` procedure.

```

$ sql -name dev
SQL> set serveroutput on
SQL> set feedback off
SQL> spool results.xml
SQL> exec ut.run(ut_junit_reporter())
SQL> spool off

```

## Measure Code Coverage to Improve Tests Over Time

The utPLSQL framework comes with a built-in code coverage reporting engine. By simply using a different test formatter, you can easily see how well your unit tests are exercising the different code paths in your PL/SQL business logic. The `ut_coverage_html_reporter()` in the example below produces an HTML page showing your unit tests' code coverage.

```

$ sql -name dev
SQL> set serveroutput on
SQL> set feedback off
SQL> spool coverage.html
SQL> exec ut.run(ut_coverage_html_reporter())
SQL> spool off

```

The output of the report for our simple two-testcase suite looks like Figure 20.

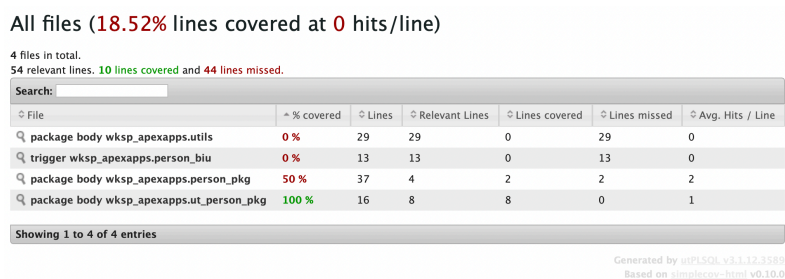


Figure 20: utPLSQL code coverage summary report

Clicking on the name of a PL/SQL program unit in the code coverage report like the person\_pkg package, you can see exactly which lines the unit tests have exercised and more importantly which lines have *not* been covered. You can see by the grey highlight in Figure 21 that no unit test invoked the validate\_integer() function. This identifies an opportunity to add a new testcase and improve the overall code coverage for your application.

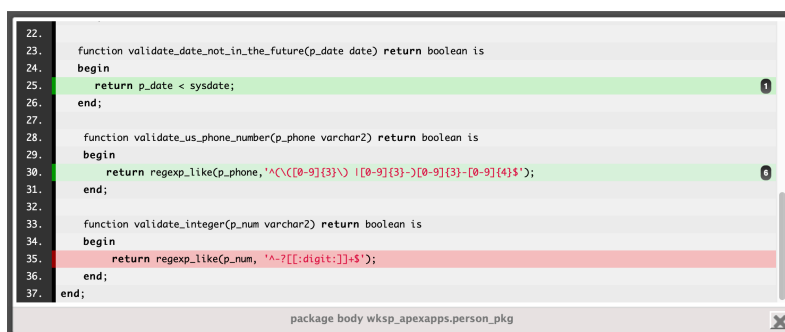


Figure 21: utPLSQL code coverage detail line coverage for a program unit

## Exercise End to End User Interface Tests with Cypress

Internal teams at Oracle building Oracle APEX applications find the free, open-source Cypress UI testing tool a productive way to create repeatable, end-to-end tests of their application user interfaces. It supports popular modern browsers like Chrome, Edge, Firefox, and Electron, and offers intuitive inspection tools to help developers quickly write and debug tests. Your Cypress tests complement your unit tests as part of a quality assurance check.

### Author UI Tests Using English-Like Commands

Cypress provides a rich set of built-in functions that make test scripts easy to write and read. Using English-like functions, your tests visit web pages, interact with the UI components on those pages, and make assertions about what should have happened after doing that. Developers love Cypress because it contains automatic “wait and retry” behavior that dramatically simplifies writing web application tests, where often the timing of interactions with the server can be unpredictable.

Developers use their favorite code editor like Visual Studio Code to author test scripts. VS Code provides excellent code-completion support for quickly entering the Cypress commands, and Cypress provides multiple ways to create reusable commands to avoid repeating yourself for commonly used steps. The test shown in Figure 22, visits a simple APEX application named HR by providing its URL, reads the username and password from environment variables, types those into the fields of the APEX login page, then clicks the *Login* button. After ensuring the browser changed to the /home page as expected, the test types a multi-word search blake Manager KING into the Interactive Report search box and presses [Enter] to perform the search. It validates the search worked as expected by checking the region before and after the search for text like BLAKE and ADAMS. Before the search, both should be present in the page but after the search only BLAKE should remain.

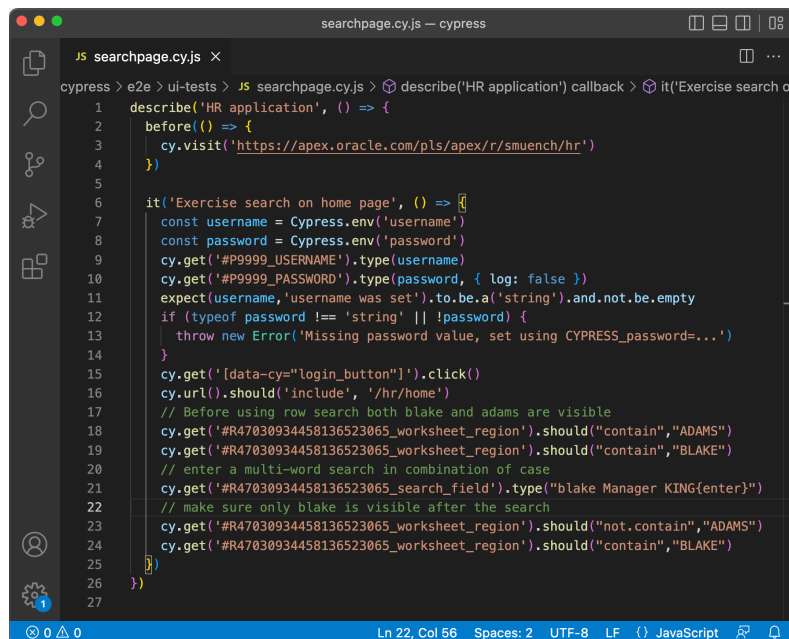


Figure 22: Simple Cypress UI test code in Visual Studio Code

Notice that all the Cypress commands start with `cy` and follow a pattern: call `cy.get()` to access a UI component by id or locator, then use functions like `type()` or `click()` or `select()` to interact with the component like an end-user would do. Assertions about what should have happened used the `should()` function and those about values expected use the `expect()` function. This lets the test read almost like an English paragraph.

```
it('Exercise search on home page', () => {
  const username = Cypress.env('username')
  const password = Cypress.env('password')
  cy.get('#P9999_USERNAME').type(username)
  cy.get('#P9999_PASSWORD').type(password, { log: false })
  expect(username, 'username was set').to.be.a('string').and.not.be.empty
  cy.get('[data-cy="login_button"]').click()
  cy.url().should('include', '/hr/home')
  // Before using row search both blake and adams are visible
  cy.get('#R47030934458136523065_worksheet_region').should("contain", "ADAMS")
  cy.get('#R47030934458136523065_worksheet_region').should("contain", "BLAKE")
  // enter a multi-word search in combination of case
  cy.get('#R47030934458136523065_search_field').type("blake Manager KING{enter}")
  // make sure only blake is visible after the search
  cy.get('#R47030934458136523065_worksheet_region').should("not.contain", "ADAMS")
  cy.get('#R47030934458136523065_worksheet_region').should("contain", "BLAKE")
})
```

## Run and Debug UI Tests Interactively

Running and debugging Cypress tests interactively is a joy as well. As shown in Figure 23, your application runs in the browser of choice. The results of each test step display in the left panel while your application runs in the right panel. Cypress provides tools to interactively select UI elements in the web page to quickly copy the right `cy.get()` command to the clipboard so you can paste it into the test you are writing in another window. It monitors the filesystem for changes so as soon as you save an updated test file, the Cypress browser window refreshes to re-run your test. As it runs, Cypress records the state of the browser at each step so you can “time-travel” back to any step in the test just by clicking on it. When you do this, you can see what the browser looked like at that step. This radically simplifies debugging tests that are not working as expected. Finally, while running



the test interactively is best for test *authors*, you can also easily run Cypress test suites as part of an automated process to incorporate both your utPLSQL unit tests and Cypress end-to-end UI tests in quality assurance checks.

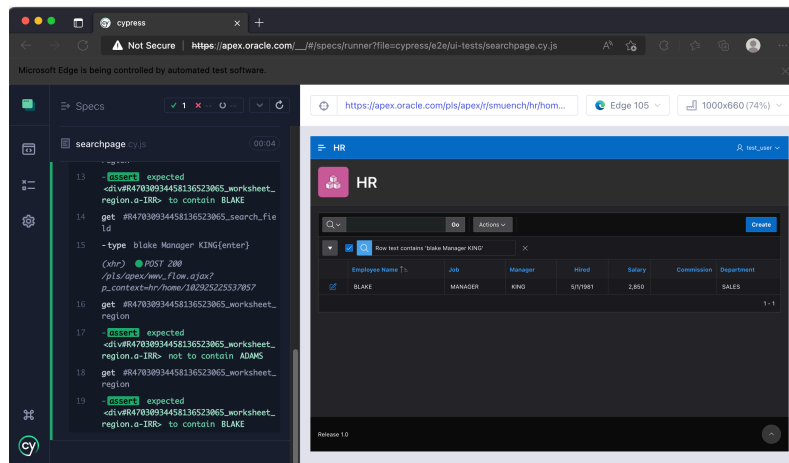


Figure 23: Running Cypress UI test interactively in Edge browser on Mac

## Release an Installable Application Artifact to Deploy Elsewhere

During the development cycle of your app's next release, team members add new features and fix bugs using working copies as they complete their assigned issues for the current milestone. At meaningful intervals, someone on the team will:

- Create a milestone branch in Git, and
- Run SQLcl's **project export** command

to export the fixes and features that teammates have merged back into the main application to add and commit the team's progress to this milestone branch in the permanent team repository. This same team member will then also run SQLcl's **project stage** command to generate corresponding distribution scripts to add and commit to that milestone branch as well.

At the same time, after merging the milestone's changes to the main branch, teams typically release an installable snapshot of their app with a new incremental version number and upload an installable artifact to a well-known location on a server. This lets other colleagues easily download a particular released version and install it later into another environment for additional interactive testing. It's also customary to tag the artifacts in the repository with a release label at the end of a milestone when a final build passes all testing and gets delivered to end-users.

### Create a Project Release

Use the SQLcl **project release** command to create a release of your project from the main branch just after merging the latest milestone branch to main:

```
$ cd yourproject
yourproject $ git status
On branch main
yourproject $ sql -name dev
SQL> project release -version 1.0.3-15
```

SQLcl confirms the successful creation of a release with:

```
| Process completed successfully
```

This release command moves files from your project's `./dist/releases/next` directory to a new `./dist/releases/1.0.3-15` directory, preparing the now-empty `./dist/releases/next` directory to accumulate the next milestone's changes.

Finally, add and commit these released changes to the main branch:

```
| yourproject $ git add --all  
| yourproject $ git commit -m "release 1.0.3 build-15"
```

Optionally, you can also tag the release in Git using the command:

```
| yourproject $ git tag release-1.0.3-15
```

### Generate a Project Release Artifact

Just after creating a formal release on the main branch, use SQLcl's **project gen-artifact** command to generate an installable artifact for the release.

```
| $ cd yourproject  
| yourproject $ git status  
| On branch main  
| yourproject $ sql -name dev  
| SQL> project gen-artifact -version 1.0.3-15
```

SQLcl confirms the successful creation of a release artifact with:

```
| Your artifact has been generated yourproject-1.0.3-15.zip
```

**NOTE:** You can also use **project gen-artifact** to create installable artifacts between releases. The command accepts **-name**, **-version**, and **-force** flags to give you control over the initial part of the archive name, the version part of the archive name, and whether it will overwrite an archive of the indicated name/version if it already exists. You can pass any values for the **-name** and **-version** flags. If you omit them, SQLcl defaults their value from your project configuration information.

### Deploy Artifact of Your App to Test or Production Environment

Whenever necessary, you can use SQLcl's **project deploy** command to deploy a release artifact of your APEX application into a target environment. Assuming you have defined a connection named **test** for your testing environment, and that you have **yourproject-1.0.3-15.zip** in the current directory, you can deploy release 1.0.3-15 to your test environment using the following commands:

```
| $ sql -name test  
| SQL> project deploy -file yourproject-1.0.3-15.zip
```

This command extracts the application artifact and runs its **./dist/install.sql** script. That performs the Liquibase update to apply appropriate distribution scripts that have not yet been applied in the current database.

Assuming you have defined a connection named **prod** for your production environment, and that you have **yourproject-1.0.3-15.zip** in the current directory, you can deploy release 1.0.3-15 to your production environment using the following commands:

```
| $ sql -name prod  
| SQL> project deploy -file yourproject-1.0.3-15.zip
```

### Aspire to Maintain Separate Environments for Dev, Test, and Prod

Developers want to work without worry of disrupting people in production or teammates doing testing. Whenever possible, isolating the development, testing, and production environments simplifies the lives of every stakeholder. With separate **DEV**, **TEST**, and **PROD** APEX *instances*, each can have the same workspace name, workspace id, and schema to keep things simple. When that's not possible or not practical, then in a single APEX instance you can still use separate *workspaces* with distinct database schemas to achieve the same benefit.

### Why Separate Environments Are a Best Practice

Many teams have members who focus on quality assurance (QA). While developers typically write the unit tests, often it's their QA colleagues who do interactive testing and write Cypress tests to validate end-to-end use cases.

It can be frustrating for testers to work in the same environment where active development occurs. End users of your team's app are even *less* forgiving of hiccups. Your app is a tool they must use to get their job done.

By giving developers, testers, and end users their own dedicated environment, each group stays most productive. Developers can iterate until their features or bug fixes are ready for testing. Testers can focus on finding problems with a recent stable build of the developers' in-progress milestone. And end users can keep getting their job done until the QA team decides the latest milestone's features and fixes meets or exceeds their quality bar.

You can deploy an APEX application to another environment using the Remote Deployment facility in APEX Builder using just a few clicks. Alternatively, you can export an application in the Builder and import it into another APEX workspace interactively. However, in this paper we focus on a command-line approach that lends itself more easily to automation.

## Use Environment Banners to Avoid Inadvertent Mistakes

Regardless of the multi-environment approach you choose, assign a distinctly colored banner to each environment so developers are always aware of which environment they're using. A green banner helps communicate that changes are allowed in the DEV environment. In the TEST environment, an ice blue banner helps convey that features are frozen there. A red banner in the PROD environment reminds developers that no changes are allowed there. Figure 24 shows this idea applied to three workspaces named `apex_dev`, `apex_test`, and `apex_prod` in a single APEX instance.

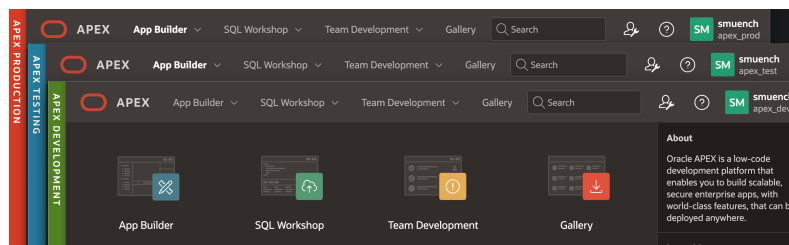


Figure 24: Separate DEV, TEST, and PROD environments using distinct banners

As shown in Figure 25, to configure your workspace's environment banner, click the *Administration* icon followed by *Manage Service > Define Environment Banner*. You can choose the banner text, color, and whether it appears in a stripe across the top or along the left side of the window.

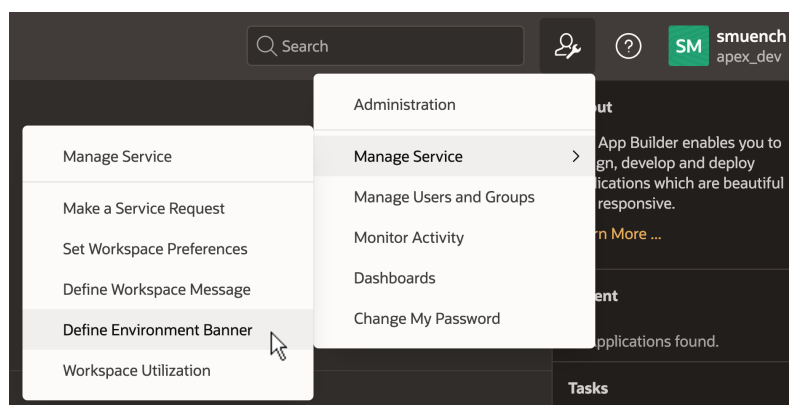


Figure 25: Defining an environment banner

## Dev, Test, and Prod in Separate Workspaces in an APEX Instance

In one APEX instance you can create multiple workspaces, and you can associate a distinct database schema with each workspace. So, consider setting up your DEV, TEST, and PROD environments as shown in Figure 26 if you have a single APEX instance to work with.



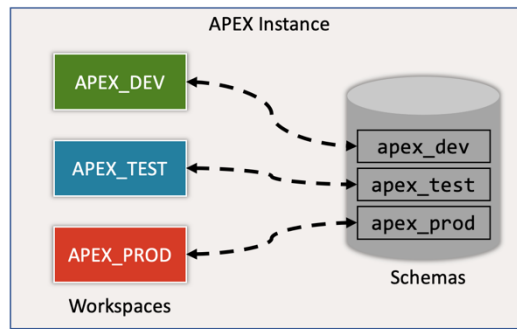


Figure 26: Three workspaces for DEV, TEST, and PROD in a single APEX instance

In this setup, developers work in the APEX\_DEV workspace. At meaningful milestones someone on the team runs **project export** to export the team's shared progress to a Git work area into a milestone branch and commit it to the permanent repository. They follow this by running **project stage** to generate the corresponding distribution scripts into the milestone branch and commit them. Then they merge the milestone changes to the main branch. When ready, using the main branch they create a release with **project release** and generate a release artifact with **project gen-artifact** to store in the location where the team keeps their release archives.

When the QA team is ready to install the latest build for testing, a team member downloads the release artifact zip file from this location and installs it into the APEX\_TEST workspace using **project deploy**. Assume that in the APEX\_DEV workspace the application in question has id 500 and an application alias of "greatapp". Since the application id and alias must be *unique* across the entire APEX instance, it's not possible to install it into the APEX\_TEST workspace with the same app id and alias. Choose a new application id like 1500 and a new alias like greatapptest and save these alternative values in a `./dist/utls/properties/test.properties` file in your project directory. The contents will look like this:

```
apex.500.workspace:APEX_TEST
apex.500.appid:1500
apex.500.schema:APEX_TEST
apex.500.alias:greatapptest
```

Similarly, to prepare for the eventual deployment to your production workspace, you'll need to invent an appropriate application id and alias for the *production* version. You'll save the production values into a `./dist/utls/properties/prod.properties` file in your project directory. The contents will look like:

```
apex.500.workspace:APEX_PROD
apex.500.appid:2500
apex.500.schema:APEX_PROD
apex.500.alias:greatapprod
```

**NOTE:** If your project includes multiple APEX applications, then your `test.properties` and `prod.properties` files can include additional similarly formatted lines whose property names would reference the other development-environment application ids. For example, in addition to the line containing `apex.500.appid:2500` your properties file would contain a line related to application 501's overridden appid of 2501 with the syntax `apex.501.appid:2501`

## One-Time Project Configuration to Support Test and Prod Deployment to Different Workspaces

With the `test.properties` and `prod.properties` files in place in your project directory, they will be included in any artifact you generate. You'll need to make a one-time change to your project's `./dist/install.sql` file to modify the following line:

```
| lb update -log -changelog-file releases/main.changelog.xml -search-path "."
```

...to add the following additional `-defaults-file &DEFAULTS_FILE` argument to the end:

```
| lb update -log -changelog-file releases/main.changelog.xml -search-path "." -defaults-file &DEFAULTS_FILE
```

Once you have modified `./dist/install.sql` in this way and committed the new properties files and this change to your main branch in Git, then to deploy any subsequent artifact you'll use the following steps.

To deploy to your test environment:

```
| $ sql -name test
SQL> DEFINE DEFAULTS_FILE=utils/properties/test.properties
SQL> project deploy -file yourproject-1.0.3-15.zip
```

Similarly, to deploy to your production environment:

```
| $ sql -name test
SQL> DEFINE DEFAULTS_FILE=utils/properties/prod.properties
SQL> project deploy -file yourproject-1.0.3-15.zip
```

## Dev, Test, and Prod in Separate APEX Instances

You can *further* simplify your life by using *separate* APEX instances for your Dev, Test, and Prod environments. This lets you maintain the same application id and alias everywhere, and as shown in Figure 27, each workspace can use the same name, workspace id, and same schema name. Of course, the distinctly colored environment banners are still a great idea to remind developers which environment they are working in.

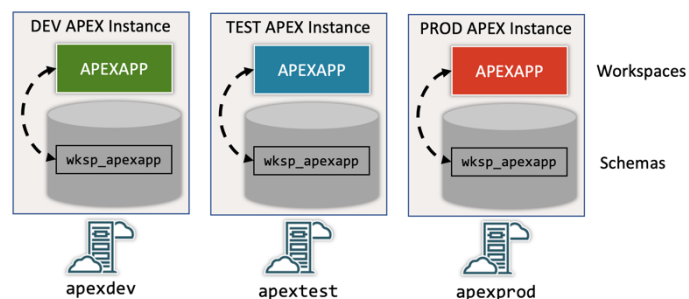


Figure 27: Three APEX instances for DEV, TEST, and PROD environments simplifies deployment

This lets you install a build into your Test environment using the command:

```
| $ sql -name test
SQL> project deploy -file yourproject_1.0.3-15.zip
```

Using the same command, you install a build into Production as well, changing only the connection name:

```
| $ sql -name prod
SQL> project deploy -file yourproject_1.0.3-15.zip
```

## Inspect Code for Performance, Security, or Quality Problems

As part of your team's development process, consider adopting some of the tools mentioned in this section to inspect your application for performance, security, or quality issues. While a deep dive into each tool's specifics is beyond the scope of this paper, it's important to be aware of the resources available so you can evaluate them for use in your own project.

## Use APEX Advisor to Run Built-in Audits

The APEX Advisor is part of the APEX Builder on the application-level *Utilities* page. It can perform 36 different audits on your APEX application to highlight various kinds of errors, security concerns, potential performance issues, and violations of best practices in the areas of usability, accessibility, and more. You can select which audits to apply, and it produces an interactive list of issues so you can easily navigate to each place in your application where it has highlighted a potential problem to correct.

## Extract SQL and JavaScript Embedded Code for Your App

You can edit the `./dbtools/project.config.json` file to add `EMBEDDED_CODE` to the list of APEX export types in the `apex > exptype` property as shown below:

```
:
"apex": {
  :
  "exptype": [ "READABLE_YAML", "APPLICATION_SOURCE ", "EMBEDDED_CODE" ]
  :
```

This produces an additional `embedded_code` directory containing `*.sql` and `*.js` files for each page and component that contains SQL or JavaScript, respectively. These code-only extract files can be useful to feed into static analysis tools that do further code inspection or other kinds of SQL or JavaScript audits as part of your development lifecycle.

## Highlight SQL Injection and Poor SQL Patterns with SQLcl

You can enable SQLcl's code scanning features using the `SET CODESCAN ON` command. When this mode is enabled, it checks every statement it processes for possible SQL injection vulnerabilities as well as SQL coding patterns that lead to poor performance. The [SQL Performance Troubleshooting](#) appendix of the SQLcl documentation lists the many anti-patterns it can detect. Figure 28 shows an example of a SQLcl warning about a SQL injection vulnerability and an example of its highlighting a construct leading to poor SQL performance.

```
8 BEGIN
9  -- Following SELECT statement is vulnerable to modification
10  -- because it uses concatenation to build WHERE clause.
11  query := 'SELECT value FROM secret_records WHERE user_name='
12          || user_name
13          || '' AND service_type='
14          || service_type
15          || ''';
16  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
17  EXECUTE IMMEDIATE query INTO rec ;
18  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec );
19  END;
20 /

SQLcl security warning: SQL injection USER_NAME line 2 -> QUERY line 11 -> QUERY line 17
-----
Procedure GET_RECORD compiled
SQL> set codescan on
SQL> select *
2  from emp
3  where upper(ename) = 'KING'
4* /

SQL performance check warning (3,6): Function calls were detected in WHERE clause predicates
-----
EMPNO ENAME      JOB          MGR HIREDATE          SAL      COMM      DEPTNO
-----
7839 KING        PRESIDENT    ----- 17-NOV-81          5000      ----- 10
```

Figure 28: Two examples of warnings using `SET CODESCAN ON` in SQLcl

## Perform Security and Quality Audits with SonarQube and ApexSec

SonarQube is a free, open-source framework that enables a flexible set of static analysis rules to check the quality and security of code in over 25 different programming languages. It can be run interactively or incorporated into build automation jobs. Any future static code analysis features from Oracle will aim to be compatible with the widely popular SonarQube ecosystem.

ApexSec is an automated security analysis product from Recx Ltd. that performs security scans of your APEX applications. It reports potential vulnerabilities and provides recommendations on the changes required to avoid them. Teams at Oracle building APEX apps use both third-party products as part of their development lifecycle.

## Understand Team-Centric vs. Feature-Centric Development

By modularizing apps, using page locking, and keeping work items focused, smaller teams can work productively on a shared APEX development instance, regularly committing team-wide progress to the Git repository. This development approach is *team-centric* since the periodic commits pushed to the repository reflect changes made by the whole team since the previous commit. In contrast, some teams prefer a *feature-centric* approach with a more formal review process for the changes made to fix a bug or add a feature. If your project needs code accountability at the feature level, APEX supports working that way, too.

### Overview of Team-Centric Development Approach

As shown in Figure 29, in a *team-centric* approach, colleagues work on assigned tasks directly in the shared APEX development environment. An issue "ticket" number like app-1234 identifies each work item and developers each use a working copy to work on their ticket before merging the changes back to the main application. Periodically, the team's progress is exported to a Git work area, committed, and pushed to the permanent team repository. Most developers on the team don't need to directly interact with the file artifacts since another team member playing the role of a build manager adds their work to Git. When appropriate, the team creates a new release and release artifact including a set of fixes and features for further testing or deployment to other environments. This approach works well for smaller teams. By design, each Git commit reflects the changes the whole team has made since the previous checkpoint. However, the more developers contributing to the team effort, the more challenging it becomes to review changes made for a particular feature or by a particular developer.

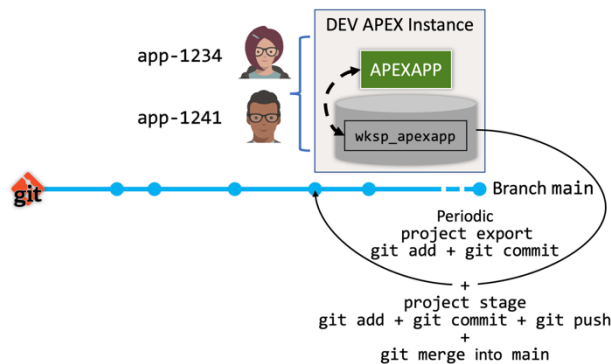


Figure 29: Team-centric development source controls regular progress of team as a whole

### Overview of Feature-Centric Development Approach

With a *feature-centric* development approach, each colleague works on one bug or feature at a time in a private work area using a private APEX instance where they are the only developer making changes. The team organizes the tasks assigned to separate team members so that colleagues avoid changing the same application artifacts while working on their respective issues. To keep team members aware of what artifacts each one is working on, teammates can use the page locking features in the central team instance. They work as long as necessary to complete their task and testing. Before submitting their changes for peer review, they update their private area to merge in any approved changes colleagues have made to application artifacts in the meantime. This proactively ensures any conflicts with their own changes get resolved. When ready, they submit their changes for peer review.

During the peer review process, colleagues can make constructive suggestions about changes required before the feature or fix is ready to be formally admitted into the application's main source code area. After addressing feedback from teammates, the developer submits her changes again for review. Eventually the changes pass inspection, are approved, and the changes get merged into the team's main source code area in the repository. That's represented in Figure 30 by the developer's branch line track rejoining the main branch line. At that time, a new build of the application gets created and installed in the central APEX development instance, which is intended to always reflect an up-to-date, runnable version of the in-development milestone.

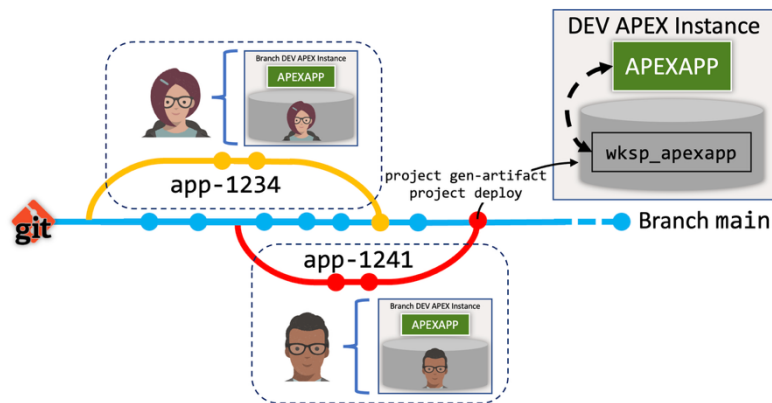


Figure 30: Feature-centric development approach teammates work on each feature in a private area

## Work Privately Using a Git Branch and "Branch Instance" of APEX

A feature-centric approach requires a private developer work area for file artifacts, and Git's ability to easily manage changes in a "branch" off the main code line delivers half of the solution. Since the web-based APEX Builder operates on an application definition installed in its runtime dictionary in the database, this approach requires pairing a Git branch with a *branch instance* of APEX. This is an APEX installation dedicated to a single developer (or tiny team) while they are working on changes for a single bug fix or feature.

## Git Manages Parallel Changes to Files Using Branches

Git's ability to easily manage parallel changes to source code files made it the world's favorite SCM system. When you initialize a Git repository for a project, it starts with one primary or "main" area in which to maintain the history of project files over time. However, with Git it's easy to create *secondary* areas where you can make experimental changes to project files in a private "sandbox". These changes don't affect the main area.

Sometimes the experiments are just that, and the secondary area just gets deleted. On other occasions, the experiments produce fruit and make sense to incorporate (or "merge") back into the main area. Git calls these independently tracked secondary areas "branches". This name evokes how a primary train line can have parallel "branch lines" on which a train can travel independently of other trains on the main line. It also suggests how a branch sprouted from a tree can grow independently of the trunk.

Branches live in the team repository for as long as necessary, so an experimental change can lie dormant and be revived in the future when the team gets time to pursue it again. Git facilitates:

- listing existing branches with **git branch**
- creating a new branch with **git checkout -b newbranchname**
- switching to work on an existing branch with **git checkout existingbranchname**
- merging changes from the main area into the current branch with **git merge main**

In contrast, merging changes from a branch back into the main area is done as the last step of the peer review process that GitLab calls a "merge request" and GitHub calls a "pull request". Often teams adopt the convention of deleting the branch at the same time its changes have been incorporated into the main area. In practice, this main area is just a branch named `main` which is special by convention since it acts as the primary location for project file version history.

When beginning new work on a feature or fix, the widely adopted convention is to create a Git branch named after the issue tracking ticket being worked on. So, for example, when beginning to work on ticket number *app-1234*, a team member creates a Git branch named `app-1234` and the project files on the branch initially reflect the current state of the files from the main branch at the time the `app-1234` branch is created. Any changes made to this initial set of files stay private to the branch.

## Branch Instance of APEX is Dedicated to a Single Feature or Fix

A "branch instance" of APEX is a private installation used by one developer (or a small group) collaborating on adding one feature or fixing one bug. Development in the branch instance **must** occur in a workspace of the same name, with the same workspace id, and same APEX version (including patch level) as the team's shared development environment. This requirement prevents uninteresting differences from polluting the file artifacts and guarantees branch file artifacts only contain changes *related* to fixing the bug or implementing the feature.

When beginning work on a new feature or fix, after creating the "feature branch" (e.g. app-1234) the corresponding branch instance of APEX needs to reflect the current state of the application definition as reflected in the main branch of the repository and the team's central APEX Development environment. There are two primary ways to setup the private branch instance:

- Clone the pluggable database (PDB) from the central team DEV environment, or
- Create a matching workspace name with identical workspace id in any Oracle database with matching version and patch level of APEX installed, and then install the latest build of the application into it

After performing one of these alternatives, the developer uses the web-based APEX Builder in their private branch instance to implement the feature or fix. At any time, they can use **project export** to export the application and its database object changes to their Git work area. When they commit and push their changes, their work is saved on a "side track" represented by branch app-1234 on the team's Git server. They follow this by a **project stage** command to generate the distribution scripts for their changes, which they add and commit to their ticket branch. When ready, the developer initiates a peer review by creating a "merge request" in Git. Once approved, the changes in the branch are merged into the main branch and officially become part of the application. The next section walks through an example.

## Walkthrough of Feature-Centric Development

To better understand a "day in the life" of a developer doing feature-centric APEX development, let's walk through a scenario where developer Gina is assigned ticket *app-1234* to implement a new feature. Figure 31 shows the high-level steps she follows to complete her assignment. In Step 1, she creates a Git branch named app-1234 to mirror the issue tracking ticket she's about to work on. In Step 2, she runs a script that clones the pluggable database of the central DEV instance and provides her the login credentials for the GINA user in the APEXAPP workspace on the branch instance where she'll do her development. In Step 3, Gina uses the APEX Builder in the branch instance to implement the feature, add her utPLSQL unit tests, add a Cypress UI test, and run all the tests to make sure everything is passing. In Step 4, Gina runs **project export** to export the file artifacts for the application and database object changes to her Git work area, then adds and commits the files to her app-1234 branch. She next runs **project stage** to generate the distribution scripts related to her changes, then adds and commits them to her branch as well. She proceeds to merge the current state of the team's main branch into her local app-1234 branch to make sure there are no conflicts with changes other teammates that have been admitted into the main branch in the meantime. In Step 5, she commits and pushes her Git branch's changes and creates a merge request to start the peer review.



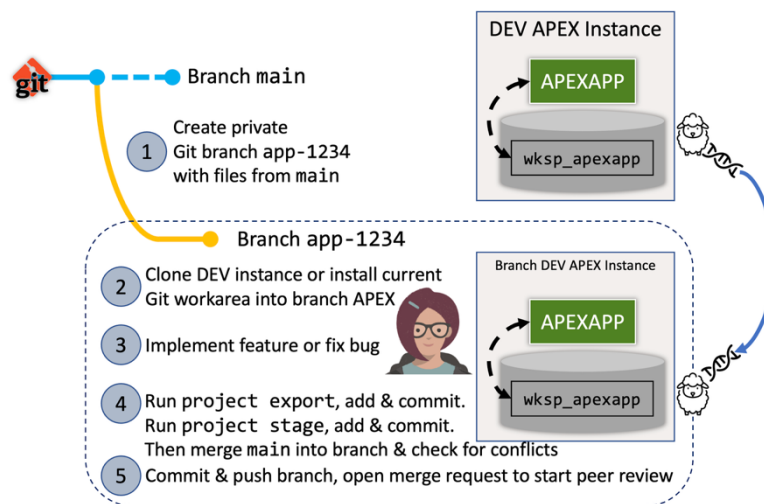


Figure 31: High-level steps for working on an APEX fix or feature in a Git branch

Two of her colleagues, Ruggero and Daisy get an email that there's a new merge request requiring peer review. They review Gina's changes and highlight some potential gotchas she should consider. After Gina addresses their feedback by repeating steps 3, 4, and 5 above, Ruggero and Daisy get an email notification about the new set of changes from Gina. Figure 32 shows the steps Daisy goes through to approve and merge Gina's changes. In Step 1, she approves the merge request after seeing Gina addressed all their concerns. She clicks a button to let Git merge Gina's changes into the main branch and delete the app-1234 branch as it's no longer needed. In Step 2, she creates a new artifact of the application based on the latest changes merged to the main branch, and in Step 3 she runs a script to install that build into the central APEX DEV instance.

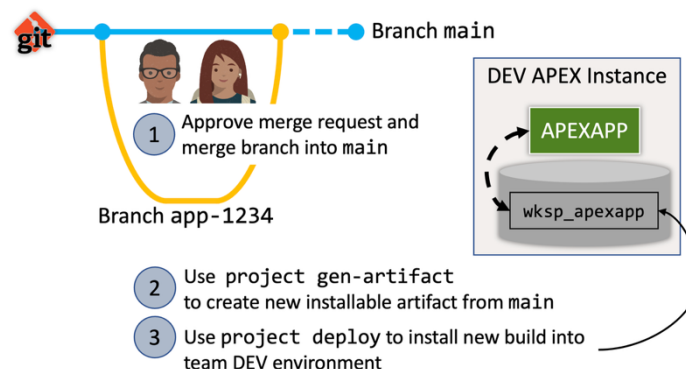


Figure 32: High-level steps involved in approving a merge request

## Committing and Pushing Changes on a Branch

Figure 33 zooms in on the concrete commands Gina ran each time she wanted to save her work in progress to the team repository. Since Git keeps her changes in branch app-1234 cleanly separated from the main branch, Gina can follow this process any number of times to checkpoint her work on the team's Git repository server. She runs the `project export` command to get the changed file artifacts from her APEX branch instance into her Git work area in `/home/gina/greatapp`, then uses `git commit` to save those changes to Git's history with a helpful comment. Then she uses `project stage` to generate the distribution files for her changes. Then, finally she uses `git push` to push the commits up to the team server.

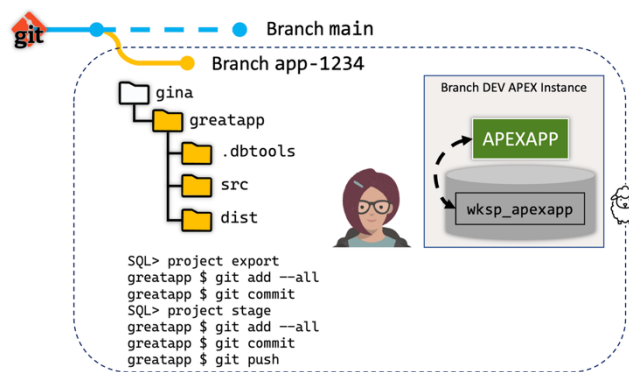


Figure 33: Exporting APEX app & DB schema changes to Git work area and generating distribution files

## Resolving Merge Conflicts

Figure 34 highlights the concrete commands Gina runs as she prepares to submit her changes for peer review. While working in the context of her branch `app-1234`, she starts by using `git commit` to save her pending changes to her local Git repository. Then she refreshes her local copy of the main branch to pull in any changes that colleagues have made in the meantime while she's been working on her feature. She does that by using `git checkout main` to switch the focus of her work area to the main branch. The `git pull` command pulls down all the latest changes on the main branch from the remote repository. Then she switches the focus of her work area back to her feature branch with `git checkout apex-1234`. Next, she explicitly merges the main branch into her local `app-1234` branch using `git merge main` to proactively check for any conflicts. APEX teammates use page locking in the central team instance to proactively avoid changing the same application artifact when working in their private "branch instance".

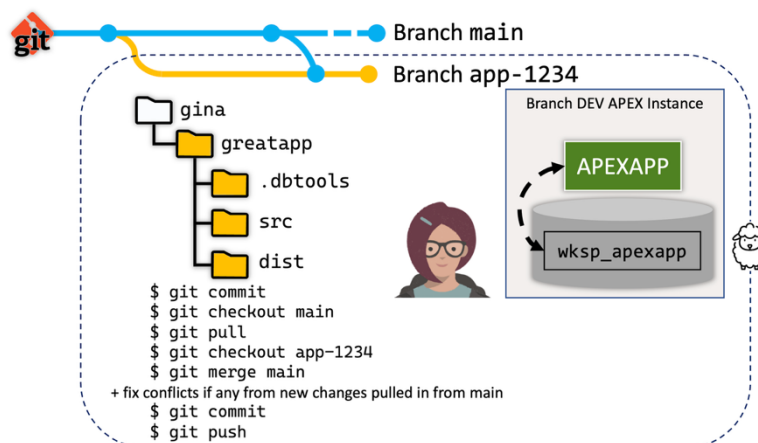


Figure 34: Merging changes from main into local branch to check for conflicts

Using the popular, free Visual Studio Code IDE, let's study three situations that might occur when merging the changes that other colleagues have made in the meantime into your private branch. Figure 35 shows VS Code's three-way merge view of a conflict for Page 3 in the APEX application. Gina's local changes appear on the right in the area titled "Yours", and the version of the same `p00003.yaml` page YAML file coming in from the team's main branch appears on the left in the area titled "Theirs". The editor shows that a team member has added a new APEX page item named `P3_SAL` in the page, while Gina has added a page item to the same Page 3 named `P3_COMM`. Git couldn't determine automatically if the overall intent was to include both page items in the page, or if only one was meant and a developer needs to decide which one is the one to keep.

In this case, let's suppose that both `P3_SAL` and `P3_COMM` should remain in the page, so Gina first ticks the checkbox at line 130 in the "Theirs" editor, then she clicks the checkbox next to line 130 in the "Yours" editor. Notice the single checkmark in the "Theirs" change and the double-checkmark in the "Yours" change. This indicates that first the "Theirs" change will be included followed by the "Yours" change to produce what's shown below: the union both changes in the resolved version of the YAML file. Gina lets VS Code know that she's done resolving the conflicts by clicking on the **Accept Merge** button in the "Result" editor below.



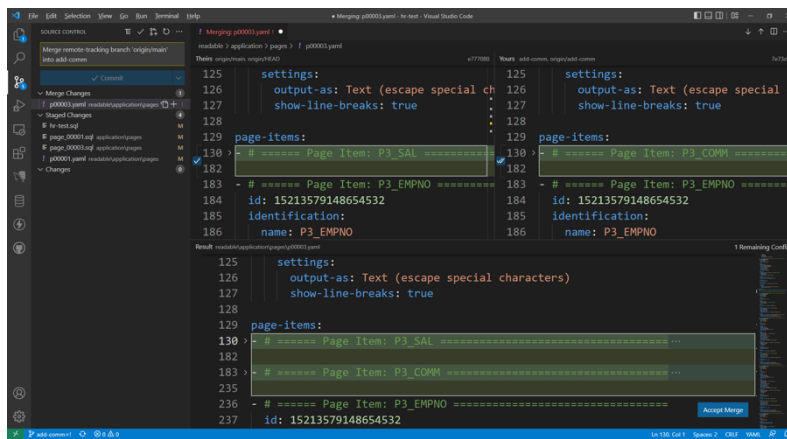


Figure 35: Resolving merge conflict by accepting both changes

Other times when resolving merge conflicts, you'll need to pick a winner for two clashing changes that can't both be included. For example, in Figure 36 we see in the "Theirs" editor that a teammate has changed the heading of the JOB column in an interactive report region on Page 3 to be "Occupation" while in Gina's local changes in her app-1234 branch she has changed the same JOB column's heading to be "Work Role". By checking only the checkbox at line 275 in the "Theirs" editor, Gina indicates that her teammates change is the "winning" one, and only the "Occupation" label remains in the resolved version of the file.

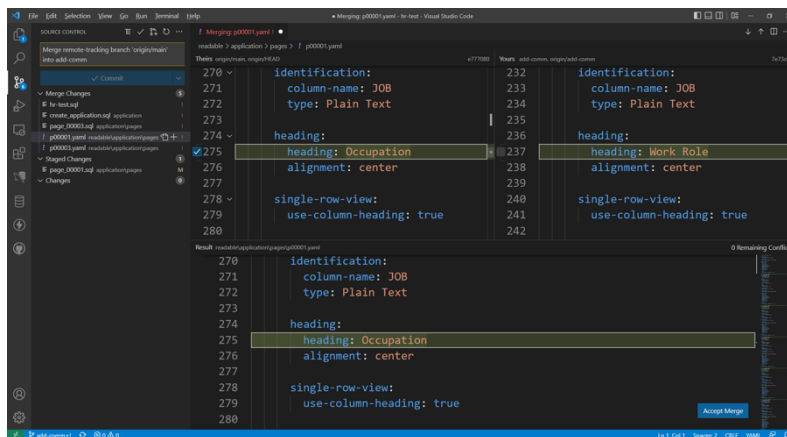


Figure 36: Resolving merge conflict by picking a winner for a clashing change

A third interesting situation that can arise during merge conflict resolution requires you to merge the elements of a delimited list. Figure 37 shows the SQL file for Page 3 and shows in the "Theirs" editor that a teammate had added the SAL column into the list of columns visible by default in the interactive report region. In Gina's local copy, she's added her COMM column into the same list as part of her feature. Since the intent is that both SAL and COMM be included in the default column list, the resolved file version editor below shows how she's carefully changed the colon-separated string value to include both SAL and COMM to resolve the conflict.

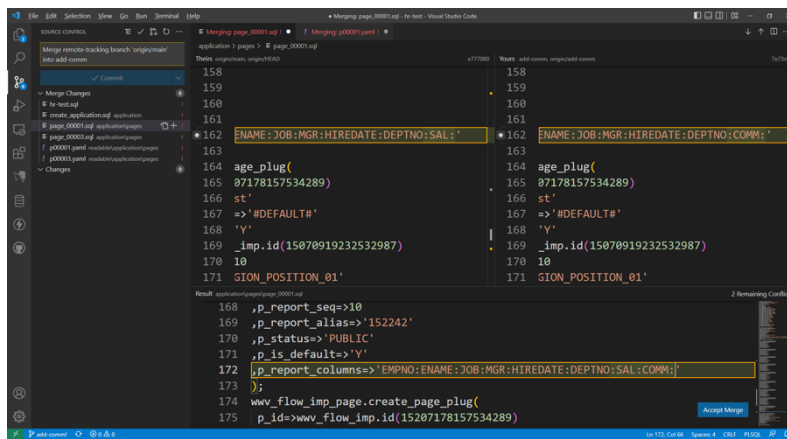


Figure 37: Resolving a merge conflict by combining delimited-list elements

## Initiating Peer Review

After Gina resolves all the conflicts and all her local test runs are passing, she commits and pushes the branch to the remote server. Then she creates a merge request to kick off the peer review process. If colleagues find issues, she addresses them in her branch, and repeats the process of:

1. Merging changes from main into your branch and addressing any conflicts
2. Running the tests to make sure all is good
3. Committing and pushing branch changes to seek merge approval or get next round of feedback

The process of creating a merge request will look different depending on what solution your team adopts for issue tracking and team collaboration, but Figure 38 shows what the process looks like using GitLab.

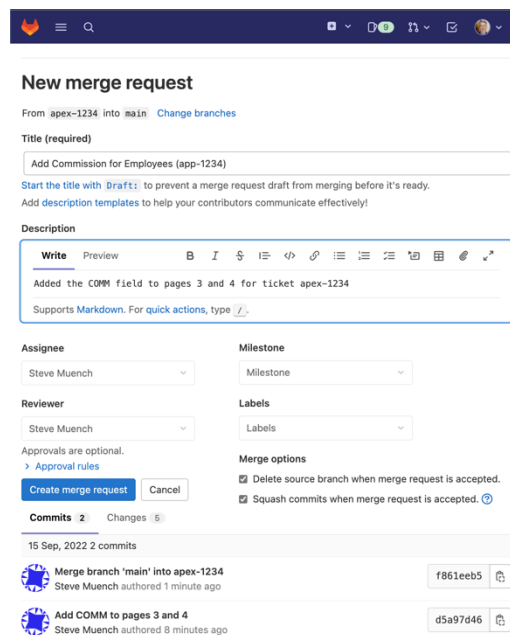


Figure 38: Creating a merge request using GitLab

## Reviewing the Merge Request

The process of reviewing a merge request is similar regardless of what solution you adopt for team collaboration. Solutions like GitHub, GitLab, Oracle VB Studio, and others let reviewers view the changes in every file artifact that pertains to the feature or fix that's been made. Reviewers can make comments related to specific lines of code very easily and carry on a threaded discussion with the developer, who may need to iterate on her changes to address the merge request review feedback. Figure 39 shows what the process looks like using GitHub, where merge requests are call "pull requests" instead.

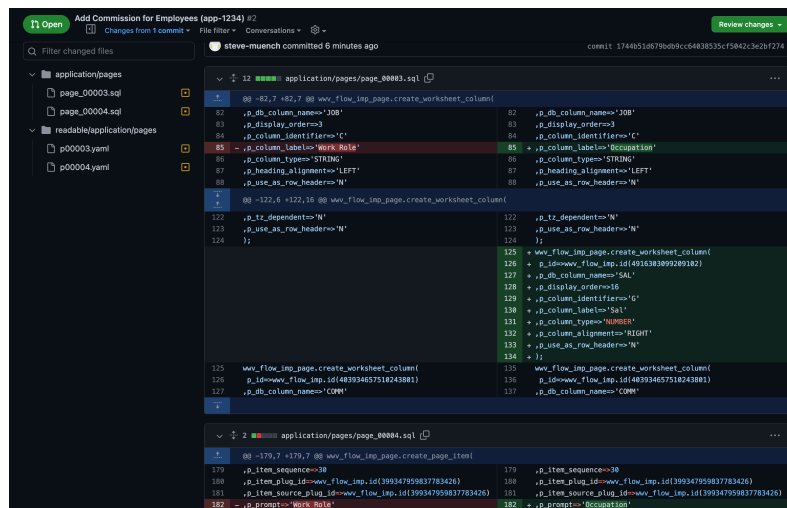


Figure 39: Inspecting changes in file artifacts while reviewing a pull request in GitHub

## Merging Approved Changes to Main

The process of approving and merging the changes pertaining to merge request is similar regardless of what solution you adopt for team collaboration. Solutions like Oracle VB Studio, GitLab, GitHub, and others let selected teammates approve the merge when the developer has addressed all open comments. The web-based interfaces of all these solutions make doing the final merge back to the main branch a one-click operation. Figure 40 shows what approving and merging a merge request looks like in Oracle VB Studio.

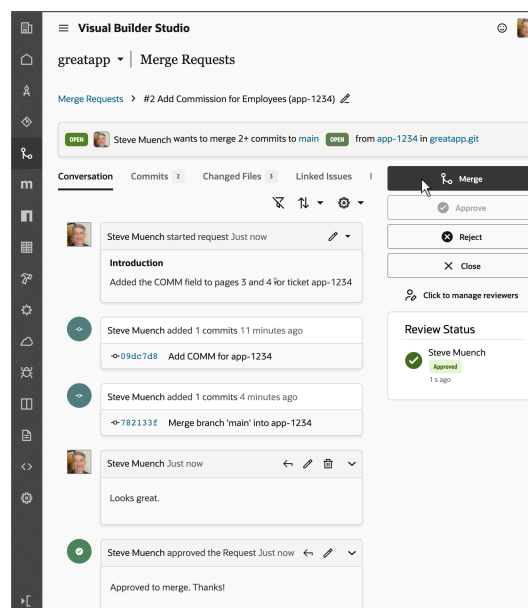


Figure 40: Approving and merging Merge Request changes with Oracle VB Studio

## Updating the Central APEX DEV Instance with the Latest Build

Once a merge request has been approved and included back into your team's main branch, you need to reflect these latest changes in the central APEX development instance. You can accomplish this with a combination of running **project gen-artifact** followed by running **project deploy**. This will ensure that the central APEX Development instance always mirrors the latest state of the main branch in the repository.

## Preparing a Release to Install in Other Environments

At some point the team will decide to issue a new release and release artifact to let their Quality Assurance team test the new group of changes in a different environment. You can accomplish this with a combination of running **project release** and **project gen-artifact**. At their convenience, the QA team can then use **project deploy** with this new release artifact to deploy to their testing environment.

## Automate the Build, Test, and Deployment Process

You've learned the building blocks to be an informed practitioner of the Oracle APEX development lifecycle. Next, you'll understand better how to put them to use in a build automation system to further simplify your daily work. It lets the changes your team commits to your Git repository automatically trigger the execution of a sequence of actions, or you can manually run an automation job whenever needed.

Below we explore two simple examples using the popular [Jenkins](#) open-source automation server. However, the concepts are applicable to similar solutions like [Oracle OCI Dev Ops](#), [GitHub Actions](#), [GitLab](#), [Oracle VB Studio](#), and others. A build automation platform helps your team become a more agile software assembly line. It lets you integrate and deliver a high-quality app, one feature and bug fix at a time, using a more predictable, continuous approach. These *continuous integration* and *continuous delivery* processes are often shortened to CI/CD.

### Understanding Automation Pipelines

In CI/CD, a *pipeline* is a sequence of actions your team needs to perform on your application artifacts. These steps can include creating release archives, running tests, performing security scans, deploying a build to a target server, sending emails, and waiting for approvals, just to name a few common activities. Since a pipeline automates actions on source code, it is usually associated with a Git repository. Your team's Git repo and your build automation server collaborate to enable the software assembly line. When a new commit adds a set of changes to the repository on the `main` or other branch, the Git server can notify the build automation server about that event. You can configure a pipeline to start automatically in response to an event like that, or at other times, you may simply run a pipeline on-demand.

A pipeline consists of one or more named *stages* that, in turn, contain specific *steps*. Jenkins queues up pipelines for execution and manages a pool of servers called build executors that carry out the work a particular pipeline requires. The first stage in most pipelines involves "checking out" a copy of an associated Git repository to create a local Git work area. This ensures the build executor has the latest copy of your application source. Other stages process the source code in a particular branch of the local Git work area. Figure 41 shows the Jenkins dashboard with two pipelines. `ExportDevToGit` is an on-demand pipeline that exports an app and database schema objects from the APEX Development environment to a branch and commits the changes, stages the project to generate the corresponding distribution files and commits the changes, merges the changes to `main`, creates a release, and finally generates the release artifact. The `greatapp` project contains a `main` pipeline triggered by a new commit on the `main` branch in that same repository. It creates a new artifact, runs the unit tests, and if they pass, deploys the artifact to the APEX Test environment so QA engineers can perform more hands-on testing. In the lower left corner of the figure, you can see there are no pipelines waiting to be executed in the build queue, and the status is *idle* of each of the two build executors.

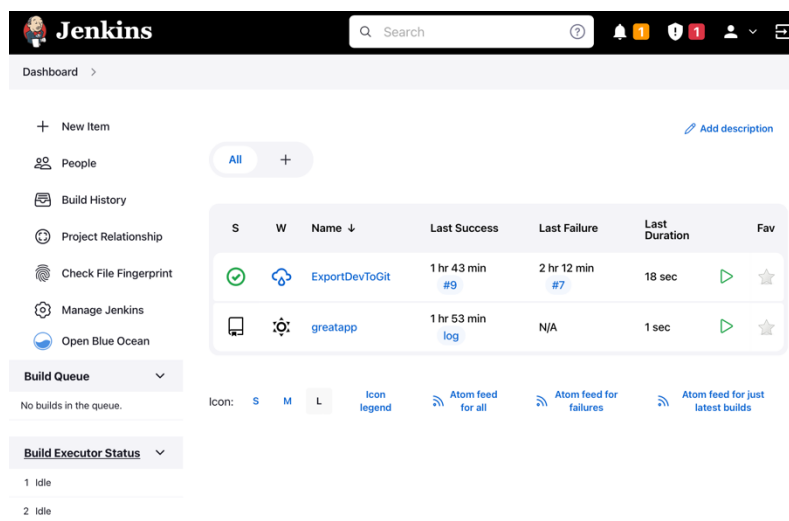


Figure 41: Jenkins dashboard showing on-demand pipeline `ExportDevToGit` and Git-triggered pipeline `greatapp`

## Pipeline to Export, Stage, and Create New Release Artifact

You define a Jenkins pipeline by creating a text file named `Jenkinsfile` in a convenient directory in your source code work area, committing it, and pushing it to the permanent repository. In the Jenkins web console, you define a new SCM-based pipeline, configure the associated Git repository URL and credentials, and mention the relative path to this new `Jenkinsfile`. In the case of the `ExportDevToGit` pipeline, the relative path name in the repository happens to be `pipelines/ExportDevToGit/Jenkinsfile`. Figure 42 shows the declarative syntax of the Jenkins pipeline `ExportDevToGit` in the Visual Studio Code editor. Notice how easy the syntax is to read. In the options section, it expresses a preference to check out the Git source code to a `greatapp` subdirectory of the root pipeline workspace directory. The environment section defines some environment variables. It's easy to see that there are two stages in the pipeline "*Setup*" and "*Export, Stage, and Create New Release Artifact*".

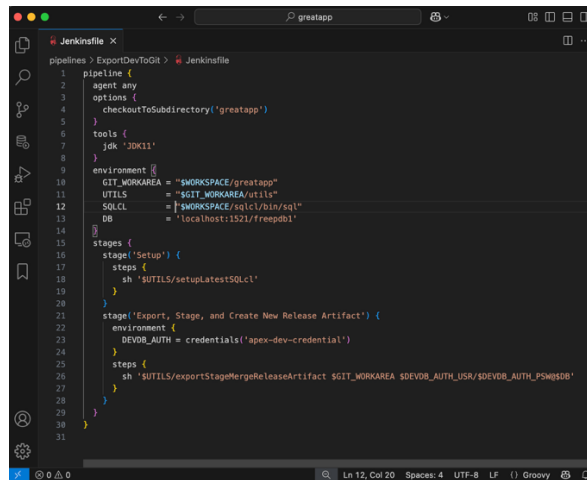


Figure 42: Declarative `ExportDevToGit` pipeline syntax in VS Code

Notice in the *ExportDevToGit* `Jenkinsfile` source code below that the two stages each has a step that runs a shell script using the `sh` command. Each step in every stage executes in the context of a pipeline workspace directory created by Jenkins on the build executor machine. The actual directory name is unimportant since you can reference it using the predefined environment variable `$WORKSPACE`. The Git repo checked out into `$WORKSPACE/greatapp` contains a `utlils` directory with small bash scripts to make the `Jenkinsfile` more readable.

In the *Setup* stage, the `setupLatestSQLcl` shell script downloads a zip file from `oracle.com` containing the latest release of `SQLcl` and unzips it in the pipeline workspace directory. In subsequent stages, anytime the pipeline needs to run `SQLcl` it can reference the `$SQLCL` variable defined in the environment section to run this handy `sql` utility from the unzipped `$WORKSPACE/sqlcl/bin` directory. In the *Export, Stage, and Create New Release Artifact* stage, we securely reference the encrypted credential for the APEX Development instance in an environment variable named `DEVDB`, and then run an appropriate `exportStageMergeReleaseArtifact` shell script that automates using `SQLcl` to:

- Create a Git branch
- Run **project export**, `git add --all`, and `git commit` on this branch
- Run **project stage**, `git add --all`, and `git commit` on this branch
- Switch to the main branch and merge this branch into main
- Run `project release` with `-version` value including `$BUILD_NUMBER` to create a new release, `git add --all`, and `git commit` to main
- Run `project gen-artifact` to create the new release artifact

```

pipeline {
  agent any
  options {
    checkoutToSubdirectory('greatapp')
  }
  tools {
    jdk 'JDK11'
  }
  environment {
    GIT_WORKAREA = "$WORKSPACE/greatapp"
    UTILS        = "$GIT_WORKAREA/utils"
    SQLCL        = "$WORKSPACE/sqlcl/bin/sql"
    DB           = 'localhost:1521/freepdb1'
  }
  stages {
    stage('Setup') {
      steps {
        sh '$UTILS/setupLatestSQLcl'
      }
    }
    stage('Export, Stage, and Create New Release Artifact') {
      environment {
        DEVDB_AUTH = credentials('apex-dev-credential')
      }
      steps {
        sh '$UTILS/exportStageMergeReleaseArtifact $GIT_WORKAREA $DEVDB_AUTH_USR/$DEVDB_AUTH_PSW@$DB'
      }
    }
  }
}

```

## Pipeline to Create Build, Run Tests, and Deploy to Test Environment

The Jenkinsfile in the root of the application's Git repository defines the main pipeline for the multi-branch pipeline project greatapp. The project's name derives from the associated Git repo. This greatapp/main pipeline creates a release and a release artifact zip file, runs the unit tests by deploying that build to an APEX instance dedicated to unit testing, and if the tests are successful, it deploys the build to the APEX Test environment. Figure 43 shows the simple visual diagram the Jenkins console provides to help you initially setup the pipeline stages and steps, but the declarative syntax is so simple that you'll likely prefer editing the pipeline in its textual format in your favorite editor. The Jenkins console has a handy *Pipeline Syntax* page that reminds you of all available steps and helps you get the correct declarative syntax for any steps you need to use, configured with appropriate options. Then you can just copy the command to paste into your pipeline file.

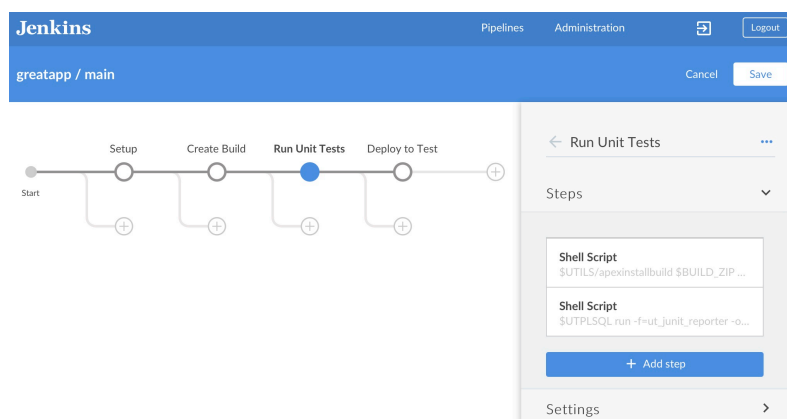


Figure 43: Visual diagram of greatapp/main pipeline to build, test, and deploy the APEX app

Every pipeline has a status page that shows the history of that pipeline's runs over time. Since the greatapp/main pipeline formats unit test results and archives the build zip file, Figure 44 shows the archived build zip is downloadable from the *Last Successful Artifacts* section at the top, a *Latest Test Result* link appears next to the clipboard icon at the bottom, and a test result trend graph appears above. The table presents the explicit and average timing of the pipeline stages.



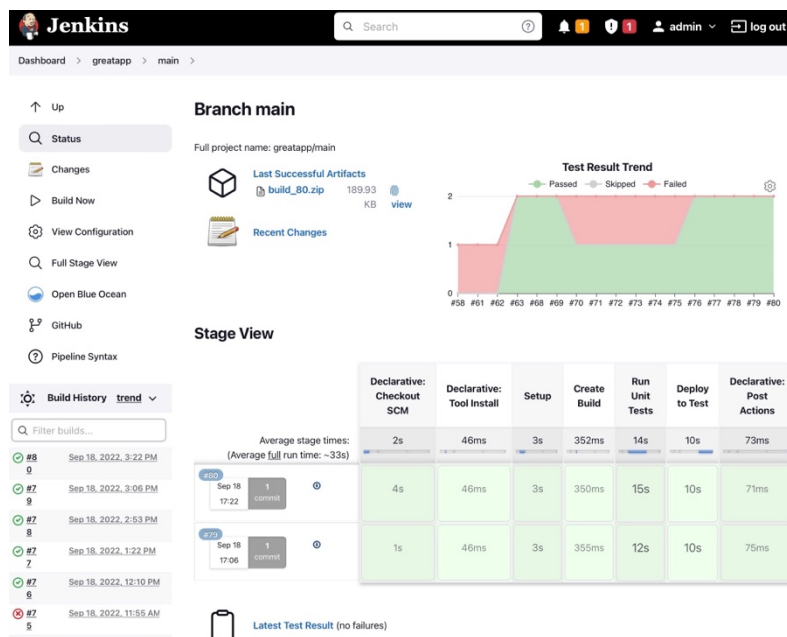


Figure 44: greatapp/main pipeline status page with downloadable build, test results link, and trend graph

The Jenkinsfile source for the greatapp/main pipeline appears below. It follows a similar pattern to the one you saw above but adds an additional step to its *Setup* stage to also download and unzip the latest version of the utPLSQL command line test runner.

- In the *Create Build* stage, it runs a genProjectArtifact shell script that accepts the \$BUILD\_NUMBER as an argument. That script uses the project gen-artifact command's -name and -version flags to create an artifact zip file named build-123456.zip.
- In the *Run Unit Tests* stage, it runs a deployProjectArtifactToUnitTest shell script that uses project deploy to deploy the generated artifact to the unit test environment. The second step in this stage runs the utPLSQL unit tests with the utplsqli command line utility, downloaded in the *Setup* stage.
- Finally, the *Deploy to Test* stage runs the deployProjectArtifactToUnitTest shell script to deploy the generated project artifact to APEX Test environment. This stage only executes if all the tests in the previous stage passed.

The pipeline ends with a post section that contains commands to archive the build zip file, format and archive the unit tests, and clean up the workspace.

```

pipeline {
  agent any
  options {
    checkoutToSubdirectory('greatapp')
  }
  tools {
    jdk 'JDK11'
  }
  environment {
    GIT_WORKAREA = "$WORKSPACE/greatapp"
    UTILS        = "$GIT_WORKAREA/utils"
    SQLCL        = "$WORKSPACE/sqlcl/bin/sql"
    UTPLSQL      = "$WORKSPACE/utPLSQL-cli/bin/utplsql"
    TESTS_DIR    = "$WORKSPACE/tests"
    TESTS_XML    = "$WORKSPACE/tests/results.xml"
    DB           = 'localhost:1521/apextrunk'
  }
  stages {
    stage('Setup') {
      steps {
        sh '$UTILS/setupLatestSQLcl'
        sh '$UTILS/setupLatestTestRunner'
      }
    }
    stage('Create Build') {
      steps {
        sh '$UTILS/genProjectArtifact $GIT_WORKAREA $BUILD_NUMBER'
      }
    }
    stage('Run Unit Tests') {
      environment {
        UNITDB = credentials('apex-unittest-credential')
      }
      steps {
        sh '$UTILS/deployProjectArtifactToUnitTest $UNITDB_USR/$UNITDB_PSW@$DB'
        sh '$UTPLSQL run -f=ut_junit_reporter -o $TESTS_XML $UNITDB_USR/$UNITDB_PSW@$DB'
      }
    }
    stage('Deploy to Test') {
      environment {
        TESTDB = credentials('apex-test-credential')
      }
      steps {
        sh '$UTILS/deployProjectArtifactToTest $TESTDB_USR/$TESTDB_PSW@$DB'
      }
    }
  }
  post {
    always {
      archiveArtifacts 'artifact/build-*/*.zip'
      junit 'tests/**/*.xml'
      cleanWs disableDeferredWipeout: true,
        patterns: [
          [pattern: 'build/*.zip', type: 'INCLUDE'],
          [pattern: 'tests/*.xml', type: 'INCLUDE']
        ]
    }
  }
}

```

Figure 45 shows the archived test results for build number 80 of the greatapp/main pipeline. Code coverage report could be included and archived just as easily.

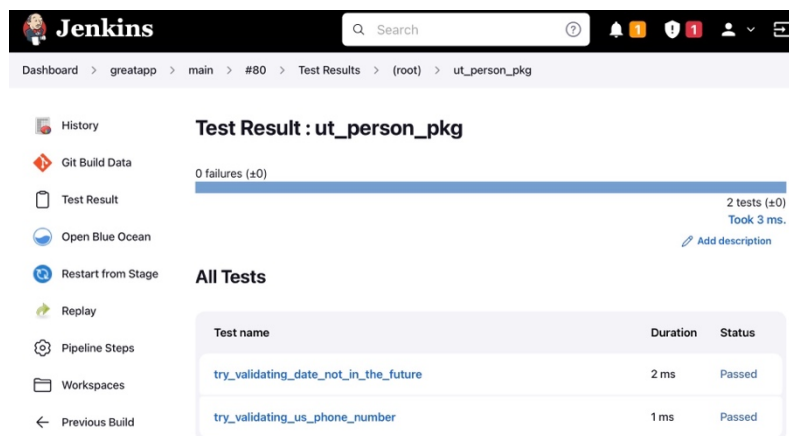


Figure 45: Formatted unit test results for a run of the greatapp/main pipeline

## Conclusion and Recommendations

Oracle APEX's industry-leading developer productivity makes it the best choice for teams of any size to build database-backed business apps for desktop or mobile users. As your end users report bugs and suggest enhancements, you and your team now have the knowhow to make steady, incremental progress against a prioritized list of these issues and ideas. With each milestone your team plans and executes, you can follow what you learned in this paper to deliver a few change requests at a time to delight your end users with consistently high-quality releases on a regular schedule. Along the way you saw examples of the APEX development lifecycle in practice using popular solutions like GitLab, GitHub, Oracle VB Studio, and Jenkins, with pointers to additional targeted solutions like Jira for issue tracking and Oracle OCI Dev Ops for build automation on Oracle Cloud.

The Oracle APEX team joins the wider APEX community in wishing you success in all the future APEX apps you and your team will build while putting your improved understanding of the development lifecycle into practice. As parting advice, consider the concrete recommendations below.

### Recommendations for Simple Apps and Small Teams

Small teams building relatively simple applications should:

- Adopt a *team-centric* approach to application development
- Track issues and plan milestones using the built-in APEX Team Development
- Work in a shared development instance, using page locking and working copies to minimize conflicts
- Embrace Git using a private repository in a free hosted service like GitHub, GitLab, or Oracle VB Studio
- Test business logic and measure code coverage with utPLSQL
- Commit team progress to Git at sensible intervals using apexexport2git or similar script
- Create a build zip of their application at the same time using apexgit2buildzip or similar script
- Aspire to use separate APEX environments for development, testing, and production
- Deploy appropriate builds to test and production as needed using apexinstallbuild or similar script.

### Recommendations for Mission-Critical Apps and Larger Teams

Teams building mission-critical applications, or larger teams in general, should:

- Adopt a feature-centric approach to application development
- Track issues in a system with Kanban boards like GitLab, GitHub, Jira, or Oracle VB Studio
- Embrace Git using a private repo in a hosted service like GitHub, GitLab, or Oracle VB Studio
- Work in private "branch instances" of APEX on Git feature branches
- Commit individual feature branch progress to Git using apexexport2git or similar script
- Conduct formal merge request code reviews with one or more teammates
- Create a build zip of their app when feature is merged to main using apexgit2buildzip or similar script
- Use separate APEX environments for development, testing, and production
- Test business logic and measure code coverage with utPLSQL
- Validate end-to-end user interface use cases with Cypress
- Inspect code for performance, security, or quality problems
- Automate build, test, and deployment using Jenkins, GitHub, GitLab, Oracle VB Studio, or OCI Dev Ops.

## Appendix A: Oracle Cloud Infrastructure Connection Considerations

As you've learned in this paper, the APEX app development lifecycle involves using the SQLcl command line tool to export APEX application definitions and database object definitions to a Git workarea. Whether you are running them interactively or as part of a build automation pipeline, the SQLcl `project export` and `project deploy` commands require a connection to the parsing schema of your team's APEX workspace. If your Oracle APEX instance runs on-premises, this is straightforward. However, there are two additional considerations when your Oracle APEX instance runs in Oracle Cloud Infrastructure.

## SQLcl Requires a Database Connection

*APEX with Autonomous Database* in Oracle Cloud Infrastructure lets external tools connect to the database. This paper's best-practice, SQLcl-based APEX development lifecycle approach works well with this environment.

The *APEX Application Development Service* supports only the HTTPS access required for low-code, browser-based development and application runtime. To export APEX application definitions to a Git workarea, you can either use the APEX Builder or the [ORDS API for APEX](#) over REST. In addition:

- *SQL Developer Web* lets you maintain database objects and ORDS REST APIs from the browser
- The `DBMS_METADATA` package lets you export schema object DDL
- The `APEX_EXPORT` package lets you export APEX application definitions
- The `APEX_ZIP` package lets you create zip files containing multiple artifacts

In short, employing a Git-based approach using *APEX Application Development Service* requires additional work.

## Connection to Database in a Private Subnet Requires a Bastion

As shown in Figure 46, if your *APEX with Autonomous Database* instance adheres to the [reference architecture](#) the database is isolated in a private subnet. To connect to the APEX workspace schema using SQLcl for development lifecycle tasks, you'll either need to [setup transit routing](#) or use a bastion. It provides restricted and time-limited access to target resources that don't have public endpoints. Learn more about bastions at the [oracle.com Bastion page](#) and in the Oracle documentation in this [Bastion Overview](#).

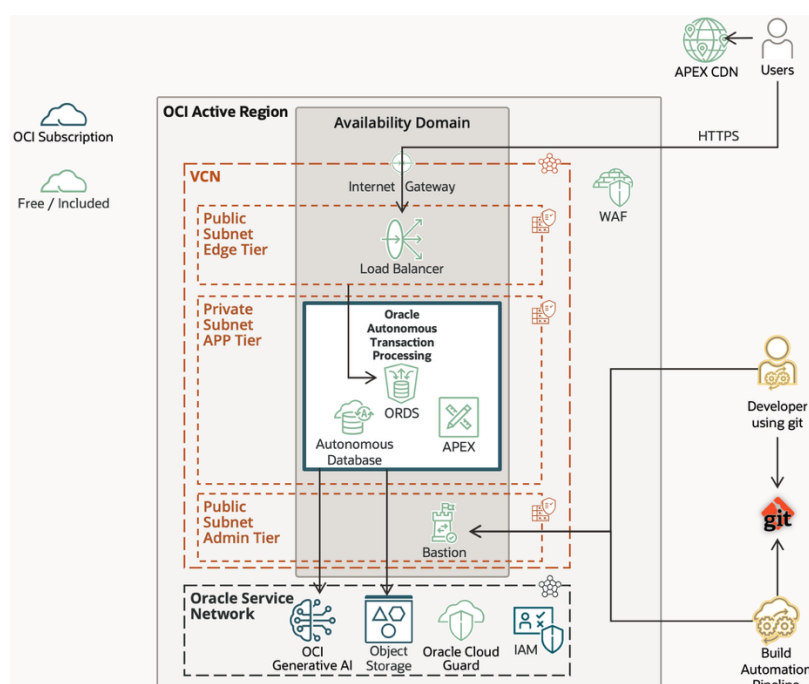


Figure 46: APEX Dev Lifecycle with APEX Reference Architecture for OCI

## Appendix B: Other Ways to Export APEX Apps and Schema Objects

The higher-level, *opinionated* SQLcl **project export** command handles the *optionally filtered* export of one or more APEX applications and database schema object definitions into a particular directory structure Oracle teams use while developing APEX applications. Your project's `.dbtools/project.config.json` file contains declarative settings that affect how the APEX applications and schema objects get exported. For completeness, if you are writing custom scripts that need to export APEX applications outside the scope of the SQLcl Projects functionality, review the [Exporting One or More Applications](#) section of the APEX Administration Guide to learn more about the lower-level **apex export** command. Refer to the [About Liquibase in SQLcl](#) section of the SQLcl guide to learn more about the lower-level commands used to generate database schema definition files.

---

### Connect with us

Call +1.800.ORACLE1 or visit [oracle.com](https://oracle.com). Outside North America, find your local office at: [oracle.com/contact](https://oracle.com/contact).

 [blogs.oracle.com](https://blogs.oracle.com)

 [facebook.com/oracle](https://facebook.com/oracle)

 [x.com/oracle](https://x.com/oracle)

---

Copyright © 2025, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

This device has not been authorized as required by the rules of the Federal Communications Commission. This device is not, and may not be, offered for sale or lease, or sold or leased, until authorization is obtained.

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

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0120

Disclaimer: If you are unsure whether your data sheet needs a disclaimer, read the revenue recognition policy. If you have further questions about your content and the disclaimer requirements, e-mail [REVREC\\_US@oracle.com](mailto:REVREC_US@oracle.com).

Author: Steve Muench, Contributors: Mike Hichwa, Salim Hlayel, Nicole Holthoefer