Implementing IAM-Based JDBC Authentication for Oracle Autonomous Database with SQL Developer

Author: Alessandro Moccia
Oracle ACE / Accenture
Independent Technical Contribution

Oracle Autonomous Database provides native integration with Oracle Cloud Infrastructure (OCI) Identity and Access Management, enabling identity-based authentication models that eliminate the use of static database credentials and significantly improve security posture.

Starting from Oracle Database 19c and later releases, Oracle JDBC drivers support token-based authentication mechanisms that allow database clients to authenticate through OCI Identity Domains and IAM-issued access tokens. When combined with SQL Developer and the ojdbc-extensions framework, this architecture enables seamless browser-based authentication and federated access models.

However, during real-world enterprise deployments, significant limitations have been observed when operating within regulated corporate networks enforcing NTLM-based outbound proxy controls.

In particular, the seamless authentication workflow implemented by the ojdbc-extensions framework does not currently provide native support for NTLM-authenticated proxies. As a consequence, browser-based authentication flows and automatic token acquisition mechanisms fail in environments where all outbound HTTPS traffic is mediated by enterprise security gateways.

These limitations have been formally acknowledged by Oracle engineering teams and documented through official support tickets and enhancement requests. Multiple defect reports and feature requests have been opened to address proxy authentication gaps both in the JDBC extension framework and in the SQL Developer authentication subsystem.

As a result, in enterprise environments subject to strict network governance, the standard seamless authentication model is often operationally unviable.

To address this architectural constraint, an alternative authentication model based on OCI CLI–driven token acquisition was designed and implemented.

By leveraging the OCI Command Line Interface, which supports explicit proxy configuration within Git Bash environments, IAM database tokens can be reliably generated even under NTLM proxy enforcement. These tokens are then securely consumed by SQL Developer and JDBC clients, enabling compliant and auditable authentication without bypassing corporate security controls.

This document presents a dual-path authentication architecture:

- a reference implementation of Oracle’s seamless authentication model for unrestricted network environments,
- and a hardened CLI-driven token-based model for proxy-constrained enterprise infrastructures.

The second model represents a practical and scalable workaround validated in production contexts and developed in collaboration with Oracle Support and engineering teams.

The implementation described herein is based on the official Oracle A-Team reference architecture:

Seamless Authentication to the Oracle Database with SQL Developer and OCI Identity Domain
https://www.ateam-oracle.com/seamless-authentication-to-the-oracle-database-with-sqldeveloper-23ai-jdbc-drivers-and-an-oci-identity-domain

and extends it with field-proven adaptations required for high-security corporate environments.

The document covers not only configuration steps, but also architectural trade-offs, security implications, and governance considerations, providing a reproducible framework for identity-centric database access in constrained enterprise networks.

The document covers:

- local environment preparation and build governance,
- secure network and proxy integration,
- identity bootstrapping using OCI CLI,
- compilation and deployment of JDBC authentication providers,
- SQL Developer runtime configuration,
- IAM DB Token lifecycle management,
- security and compliance implications,
- operational best practices.

The objective is to provide architects, senior developers, and security engineers with a reproducible, auditable, and production-ready framework for implementing identity-based database authentication in Oracle Cloud environments.

1 Step 1 – Local Environment Preparation and ojdbc-extensions Build

1.1 Introduction and Scope

This step describes the preparation of the local development environment and the compilation of the Oracle ojdbc-extensions project required to enable Identity and Access Management (IAM)–based authentication for Autonomous Database connections from SQL Developer.

The implementation follows the official Oracle A-Team guidance published in:

Oracle A-Team Blog – Seamless Authentication to the Oracle Database with SQL Developer and OCI Identity Domain
https://www.ateam-oracle.com/seamless-authentication-to-the-oracle-database-with-sqldeveloper-23ai-jdbc-drivers-and-an-oci-identity-domain

The objective of this step is to build and install the JDBC authentication providers that extend the standard Oracle JDBC driver with OCI Identity Domain and IAM Token support.

This configuration represents the foundational layer for both:

- Seamless browser-based authentication
- Manual IAM DB Token authentication via OCI CLI

1.2 Objectives of Step 1

The main goals of this step are:

- Prepare a controlled local build environment
- Install and configure Apache Maven
- Configure Java and Oracle client components
- Clone and build the `ojdbc-extensions` project
- Generate OCI-compatible JDBC authentication providers
- Install compiled artifacts in the local Maven repository

The environment is configured exclusively within Git Bash, without modifying global Windows system variables, in order to comply with corporate security policies and minimize system-level impact.

The environment is configured exclusively within Git Bash, without modifying global Windows system variables, in order to comply with corporate security policies and minimize system-level impact.

1.3 Development Environment Overview

The following components are used:

Component	Version / Location
Operating System	Windows 11 (Corporate Managed)
Shell	Git Bash (MINGW64)
Java	JDK 17 (System-level installation)
Maven	Apache Maven 3.9.11 (Local installation)
OCI CLI	Installed via official installer
Wallet	Autonomous Database TCPS Wallet

All development activities are executed inside Git Bash.

Apache Maven Installation and Build Environment Integration

Apache Maven was adopted as the reference build automation platform for compiling the `ojdbc-extensions` framework, due to its native support for multi-module Java projects, dependency governance, artifact lifecycle management, and compatibility with Oracle-provided JDBC extension architectures. To guarantee full control over toolchain versioning and ensure reproducibility of the build environment, Maven was installed manually using the official Apache binary distribution (`apache-maven-3.9.11-bin.zip`), downloaded from the Apache Software Foundation repository. The archive was extracted into a dedicated tools directory (`C:\Tools\apache-maven-3.9.11`) in order to isolate development tooling from system-managed paths and corporate-managed software inventories. Rather than modifying global Windows environment variables, Maven was intentionally exposed only within the Git Bash runtime by extending the shell PATH variable through the `.bashrc` startup configuration file. This design choice ensured strict separation between corporate-managed operating system components and locally governed development tooling, while enabling controlled execution of build pipelines within regulated enterprise environments. After environment initialization, Maven availability and runtime binding were verified to confirm proper integration with the installed Java platform and to enable deterministic execution of subsequent build processes.

Commands executed for Maven installation and validation:

```
# Download Apache Maven (from official website)
# https://maven.apache.org/download.cgi

# Extract archive
unzip apache-maven-3.9.11-bin.zip -d /c/Tools/

# Configure Maven in Git Bash environment
echo 'export PATH="/c/Tools/apache-maven-3.9.11/bin:$PATH"' >> ~/.bashrc
```

```
# Reload shell configuration
source ~/.bashrc

# Verify Maven installation
mvn -v
```

ojdbc-extensions Source Governance and Provider Compilation

IAM-based JDBC authentication for Oracle Autonomous Database is implemented through the `ojdbc-extensions` framework, which extends the standard Oracle JDBC driver with pluggable identity providers, OAuth token handlers, and OCI Identity Domain integration modules. In accordance with Oracle A-Team reference architecture, the official `ojdbc-extensions` repository was cloned from GitHub into a controlled local workspace and governed through strict version management. To ensure compatibility with SQL Developer and supported Oracle Cloud services, the validated release tag `v1.0.1` was explicitly selected, placing the repository in a detached HEAD state. This configuration was intentionally adopted to guarantee full reproducibility of the build process and to avoid instability introduced by unvalidated development branches. The project, structured as a multi-module Maven build defined by a root `pom.xml`, was then compiled to generate OCI-compatible authentication providers required for both seamless and token-based workflows. The resulting artifacts were installed in the local Maven repository and prepared for downstream integration.
The OCI IAM JDBC integration is based on the official `ojdbc-extensions` reference implementation published by Oracle. The project was obtained from the public GitHub repository and aligned to the stable `v1.0.1` release branch, in accordance with A-Team guidance. This version was selected due to its documented stability and compatibility with SQL Developer 24.x and the corresponding OCI SDK dependencies. After cloning or downloading the repository, the build process was executed from the root project directory, containing the parent `pom.xml` file and all provider submodules. Apache Maven was installed and configured as the standard build tool responsible for dependency resolution, compilation, and artifact installation. The command `mvn clean install` was executed at project root level, triggering the download of all required transitive dependencies from public repositories, including the core OCI Java SDK, HTTP client libraries, cryptographic providers, and auxiliary runtime components. During this phase, Maven populated the local repository with the compiled provider modules and all associated dependencies. In particular, the OCI JDBC resource provider relies on the OCI SDK to perform identity resolution and token validation, following the same architectural model adopted by OCI CLI. A successful build resulted in the installation of all required artifacts under the local `.m2` repository. These artifacts subsequently became the authoritative source for all runtime dependencies referenced in `product.conf`. Version alignment was treated as a critical architectural constraint, as mismatches between the built providers, OCI SDK libraries, and JDBC driver versions were observed to cause authentication and classloading failures during early integration attempts.

Commands executed for source acquisition, version governance, and build:

```
# Navigate to tools workspace
cd /c/Tools

# Clone ojdbc-extensions repository
git clone https://github.com/oracle/ojdbc-extensions.git

# Enter repository directory
```

```
cd ojdbc-extensions

# Checkout validated A-Team release
git checkout v1.0.1

# Execute multi-module Maven build
mvn clean install

# Verify repository state
git status

# Inspect installed Oracle artifacts
ls ~/.m2/repository/com/oracle
```

Java Runtime and Oracle Client Integration

The Java Runtime Environment represents the core execution layer for all JDBC-related tooling. In this implementation, Java was installed and managed at the operating system level in order to ensure version consistency, security patch governance, and compatibility with Oracle-supported JDBC drivers.

The selected runtime platform was a recent Long-Term Support release, installed system-wide and inherited by Git Bash through the Windows environment:

C:\jdk-17\bin

This approach avoids shell-specific Java overrides and guarantees consistent behavior across SQL Developer, Maven, and command-line tools.

Runtime availability was verified using:

java -version

Secure database connectivity was enabled through the Oracle Instant Client and Autonomous Database wallet. The Instant Client provided the native networking stack required for SQL*Net and TCPS operations, while the wallet supplied cryptographic material, trust anchors, and service definitions.

The wallet directory was exposed to all Oracle tooling through the following environment variable:

export TNS_ADMIN=/c/atp_environment/Wallet_ATP

This configuration activated mutual TLS authentication, encrypted transport, and secure service resolution, aligning all client connections with Oracle's Zero Trust security model.

Enterprise Network Governance and Proxy Enforcement

The development environment operated within a corporate infrastructure protected by NTLM-based outbound proxy controls. All external communications were routed through centralized security gateways responsible for traffic inspection, authentication, and compliance enforcement.

Direct Internet access was therefore unavailable, and development tools required explicit proxy configuration.

Within Git Bash, network access was enabled by exporting authenticated proxy variables:

export HTTP_PROXY=http://<proxy-host>:8080

export HTTPS_PROXY=http://<proxy-host>:8080

This configuration ensured uninterrupted connectivity for Maven dependency resolution, GitHub repository access, OCI REST API communication, and token-based authentication workflows. Without these settings, the entire build and integration pipeline would have been blocked at the network layer.

OCI CLI Installation and Identity Bootstrap

The Oracle Cloud Infrastructure Command Line Interface constitutes the primary identity bridge between the local workstation and OCI control plane services. It enables secure API signing, token generation, and profile-based authentication.

The CLI was installed using Oracle's official bootstrap mechanism:

bash -c "$(curl -L https://raw.githubusercontent.com/oracle/oci-cli/master/scripts/install/install.sh)"

Following installation, authentication profiles and signing keys were configured under the standard directory:

~/.oci

This directory contains tenancy metadata, cryptographic material, regional configuration, and user credentials. From an architectural standpoint, this step establishes the trust boundary between the workstation and OCI Identity Domains, enabling subsequent IAM DB Token operations.

Source Governance and ojdbc-extensions Acquisition

Oracle IAM-based JDBC authentication is implemented through the ojdbc-extensions framework, which extends the base JDBC driver with pluggable identity providers compliant with Oracle's authentication SPI.

These extensions implement:

- OCI Identity Domain integration,
- OAuth token handling,
- security context propagation,
- credential delegation mechanisms.

To obtain the validated implementation, the official Oracle repository was cloned into a controlled local workspace:

cd /c/Tools

git clone https://github.com/oracle/ojdbc-extensions.git

cd ojdbc-extensions

Strict version governance was applied in order to ensure compatibility with SQL Developer and supported Oracle Cloud services. The reference implementation identified by Oracle A-Team is based on the stable tag v1.0.1, which was explicitly selected:

git checkout v1.0.1

Operating in a detached HEAD state was intentional and ensured full reproducibility of the build environment, avoiding instability introduced by unvalidated development branches.

Multi-Module Build and Provider Generation

The ojdbc-extensions repository is structured as a multi-module Maven project. The root pom.xml defines a hierarchy of authentication providers, security adapters, and integration components.

Each module encapsulates a specific identity capability and is compiled into a deployable artifact.

The complete build was executed from the project root:

cd /c/Tools/ojdbc-extensions

mvn clean install

During execution, Maven resolved dependency graphs, validated artifact integrity, compiled source modules, packaged authentication providers, and installed the resulting binaries in the local Maven repository.

A successful build concluded with the standard:

BUILD SUCCESS

This outcome confirmed that all OCI-specific JDBC providers had been generated correctly and were available for downstream integration.

Post-Build Validation and Environment Integrity

Following compilation, the environment was validated to ensure structural consistency.

The active Maven installation was verified using:

which mvn

Confirming isolation within the Git Bash environment.

The presence of compiled Oracle artifacts was validated through inspection of the local repository:

ls ~/.m2/repository/com/oracle

Repository status was checked to ensure alignment with the validated tag:

git status

These verification steps provided forensic assurance that the environment was aligned with the A-Team reference implementation and suitable for production-grade integration.

Architectural Outcome of Step 1

At the conclusion of this phase, the workstation environment satisfied all structural and security prerequisites required for IAM-based JDBC authentication.

The platform provided:

- a governed Java runtime,
- an isolated Maven build pipeline,
- encrypted TCPS connectivity,
- proxy-compliant network access,
- OCI identity anchoring,
- validated authentication providers,
- reproducible source governance.

This foundation enabled deterministic integration with SQL Developer and supported both seamless browser-based authentication and manual IAM DB Token workflows

Step 2 – SQL Developer Integration (product.conf, classpath, provider loading, token resolution) and Under Enterprise NTLM Proxy Constraints.

The integration of IAM-based JDBC authentication into SQL Developer was initially designed according to Oracle A-Team reference architecture, relying on seamless token acquisition through the ojdbc-extensions framework and the OCI Java SDK.

This model assumes unrestricted outbound HTTPS connectivity for identity federation, OAuth token exchange, and service discovery.

In regulated enterprise environments enforcing NTLM-authenticated proxy gateways, this assumption does not hold.

Despite full runtime integration of authentication providers, REST clients, cryptographic libraries, and proxy parameters at JVM level, seamless authentication consistently failed during OCI token acquisition.

Root cause analysis, performed in collaboration with Oracle Support and engineering teams, confirmed structural incompatibilities between the OCI SDK HTTP stack (Apache HttpClient / Jersey) and NTLM-mediated authentication flows.

In particular, the SDK was unable to complete multi-stage proxy authentication handshakes required for secure token exchange.

These limitations were formally acknowledged by Oracle through defect reports and enhancement requests covering:

- NTLM proxy negotiation failures,
- incomplete credential propagation,
- inconsistent HTTPS tunneling behavior,
- inability to reuse authenticated proxy sessions.

As a result, the seamless authentication model proved operationally unreliable in NTLM-governed networks, even when all documented configuration steps were correctly applied.

Controlled Runtime Injection and Initial Seamless Attempt

Prior to identifying the proxy constraint, SQL Developer was configured with a fully resolved runtime environment including:

- custom IAM JDBC providers,
- OCI Java SDK modules,

- REST client infrastructure,
- cryptographic extensions,
- TCPS security libraries,
- JVM-level authentication parameters.

All required dependencies were injected through product.conf using explicit AddJavaLibFile directives, eliminating classloading ambiguity.

IAM authentication was enabled through system properties forcing OCI-based token resolution.

Enterprise proxy settings were applied at JVM level using standard networking properties.

Despite this configuration, OCI token requests consistently failed at runtime, confirming that the limitation was not caused by misconfiguration, but by architectural incompatibility.

Oracle Support Escalation and Engineering Validation

Multiple support cases were opened to analyze the failure pattern.

Packet-level traces, JVM debug logs, and OCI SDK diagnostics were collected and shared with Oracle engineering.

The investigation confirmed that the current OCI Java SDK HTTP providers do not reliably support NTLM-authenticated proxy environments for IAM token acquisition.

Oracle engineering teams formally acknowledged the limitation and registered enhancement requests to address:

- native NTLM proxy support,
- improved proxy credential handling,
- alternate authentication providers.

At the time of implementation, no certified fix was available.

Adoption of CLI-Based Token Acquisition Model

To ensure operational continuity, a secondary authentication architecture was implemented based on OCI CLI–driven token generation.

Unlike the embedded Java SDK, OCI CLI operates in a shell environment where proxy variables can be explicitly controlled and authenticated using external credential managers.

This allows OCI REST endpoints to be accessed reliably, even under NTLM enforcement.

In this model:

1. IAM database tokens are generated using OCI CLI within a proxy-aware Git Bash environment.
2. Tokens are stored securely in a controlled local directory.
3. SQL Developer is configured to consume externally generated tokens.
4. JDBC authentication proceeds without invoking SDK-based token acquisition.

This approach bypasses the SDK HTTP stack and delegates network authentication to a toolchain that natively supports corporate proxy models.

Dual-Mode Architecture: Seamless and CLI-Driven Authentication

The final architecture supports two operational modes:

Mode 1 – Seamless Authentication (Unconstrained Networks)

Used in environments without NTLM proxy mediation.

Token acquisition is performed directly by the JDBC providers through the OCI SDK.

Mode 2 – CLI-Driven Token Authentication (Enterprise Networks)

Used in regulated environments with NTLM enforcement.

Token acquisition is performed externally via OCI CLI.

SQL Developer operates in token-consumer mode.

This dual-mode design ensures portability across heterogeneous enterprise infrastructures.

Reference Configuration (product.conf – Extract)

```
################################################################################
################################################################################
# SQL Developer product.conf
# Purpose: Extend SQL Developer JVM to support OCI IAM authentication
#       using externally generated database tokens.
################################################################################


# ------------------------------------------------------------------------
# Default Oracle header
# Describes syntax and formatting rules.
# No functional impact on IAM configuration.
# ------------------------------------------------------------------------


################################################################################
# JDK Configuration (optional)
################################################################################
# If uncommented, forces SQL Developer to use a specific JDK.
# In this project, JDK is inherited from system installation.
# No override required.
#
# SetJavaHome /path/jdk


################################################################################
# JVM Memory Configuration (optional)
```

```
################################################################################
########
# Heap sizing options.
# Left commented because default sizing was sufficient
# and not related to IAM integration.
#
# AddVMOption -Xms128m
# AddVMOption -Xmx800m

################################################################################
#####
# SQL Developer – OCI IAM JDBC AUTH EXTENSIONS
################################################################################
#####
# This block represents the custom runtime layer added
# to transform SQL Developer into an IAM-aware client.
# All following directives override default classpath behavior.

# ========================================================================
# Oracle JDBC IAM Providers (from ojdbc-extensions v1.0.1 build)
# ========================================================================

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc-provider-common/1.0.1/ojdbc-
provider-common-1.0.1.jar
# Base SPI and shared authentication framework.
# Required for any external authentication provider.
AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc-provider-oci/1.0.1/ojdbc-provider-oci-
1.0.1.jar
# OCI-specific IAM implementation.
# Handles token parsing, validation, and identity binding.
# Without this module, IAM authentication is impossible.

# ========================================================================
# OCI Java SDK Core Modules
# ========================================================================

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-common/3.37.0/oci-java-sdk-
common.jar
# Base OCI SDK types: auth, region, config, signing.
```

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-common-httpclient/3.37.0/oci-java-sdk-common-httpclient.jar

HTTP abstraction layer used by SDK.

#AddJavaLibFile ... common-httpclient-jersey.jar

Disabled: Jersey-based client unstable behind NTLM.

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-identitydataplane/3.37.0/oci-java-sdk-identitydataplane.jar

IAM / identity services.

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-workrequests/3.37.0/oci-java-sdk-workrequests.jar

Async OCI operations support.

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-circuitbreaker/3.37.0/oci-java-sdk-circuitbreaker.jar

Fault tolerance / retry framework.

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-database/3.37.0/oci-java-sdk-database.jar

Database service client.

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-shaded-full/3.52.1/oci-java-sdk-shaded-full.jar

Shaded dependencies to avoid class conflicts.

Added after repeated dependency resolution issues.

==

Oracle JDBC Driver Override

==

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc11/<VERSION>/ojdbc11.jar

Overrides SQL Developer embedded driver.

Required for full IAM + Java 21 compatibility.

==

JAX-RS / Jersey REST Runtime

==

AddJavaLibFile <M2_REPO>/javax/ws/rs/javax.ws.rs-api/2.1.1/javax.ws.rs-api.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-common/2.35/jersey-common.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-client/2.35/jersey-client.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-apache-connector/2.35/jersey-apache-connector.jar

AddJavaLibFile <M2_REPO>/org/glassfish/hk2/hk2-api/2.6.1/hk2-api.jar

AddJavaLibFile <M2_REPO>/org/glassfish/hk2/hk2-utils/2.6.1/hk2-utils.jar

AddJavaLibFile <M2_REPO>/javax/inject/javax.inject/1/javax.inject.jar

REST client stack required by OCI SDK.

Added after NoClassDefFoundError on javax.ws.rs / jersey.

SQL Developer does not ship a compatible REST runtime.

==

Cryptography (BouncyCastle)

==

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcprov-jdk15to18/1.74/bcprov.jar

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcpkix-jdk15to18/1.74/bcpkix.jar

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcutil-jdk15to18/1.74/bcutil.jar

Required for PEM key parsing and request signing.

Fixes failures during private key loading.

==

Resilience and Dependency Support

==

AddJavaLibFile <M2_REPO>/io/github/resilience4j/resilience4j-core/1.7.1/resilience4j-core.jar

AddJavaLibFile <M2_REPO>/io/github/resilience4j/resilience4j-circuitbreaker/1.7.1/resilience4j-circuitbreaker.jar

AddJavaLibFile <M2_REPO>/io/vavr/vavr/0.10.2/vavr.jar

Required by OCI SDK fault tolerance layer.

Added after runtime crashes under transient network failures.

==

Logging

==

```
AddJavaLibFile <M2_REPO>/org/slf4j/slf4j-api/1.7.36/slf4j-api.jar
AddJavaLibFile <M2_REPO>/org/slf4j/slf4j-simple/1.7.36/slf4j-simple.jar

# Enables correlated logging across JDBC, SDK, SSL, REST layers.

# ========================================================================
# IAM Activation (JDBC Driver Level)
# ========================================================================

AddVMOption -Doracle.jdbc.tokenAuthentication=OCI
# Forces token-based authentication.

AddVMOption -Doracle.jdbc.iam=true
# Enables IAM mode.

AddVMOption -Doracle.jdbc.iam.auth=OCI_IAM
# Selects OCI IAM provider.

AddVMOption -Doracle.jdbc.iam.config.file=<OCI_CONFIG>
# Points to ~/.oci/config.

AddVMOption -Doracle.jdbc.iam.profile=DEFAULT
# Selects OCI profile.

AddVMOption -Doracle.jdbc.tokenLocation=<TOKEN_PATH>
# CRITICAL: Path where OCI CLI writes DB token.
# JDBC reads token from here.
# No runtime acquisition.

AddVMOption -Doracle.jdbc.debug=true
# Enables driver diagnostics.

# ========================================================================
# Wallet / TCPS Support
# ========================================================================

AddJavaLibFile <INSTANT_CLIENT>/ojdbc11.jar
AddJavaLibFile <INSTANT_CLIENT>/oraclepki.jar
```

```
AddJavaLibFile <INSTANT_CLIENT>/osdt_core.jar
AddJavaLibFile <INSTANT_CLIENT>/osdt_cert.jar

# Enables PKI, wallet parsing, TCPS handshake.

AddVMOption -DTNS_ADMIN=<WALLET_DIR>
# Standardizes wallet and service resolution.

# ========================================================================
# HTTP Client Selection
# ========================================================================

AddVMOption -Doracle.bmc.sdk.useApacheClient=true
# Forces Apache HTTP client.

# ========================================================================
# Proxy Configuration (Documented but Ineffective)
# ========================================================================

#AddVMOption -Djava.net.useSystemProxies=true
#AddVMOption -Dhttp.proxyHost=...
#AddVMOption -Dhttp.proxyPort=...
#AddVMOption -Dhttp.auth.ntlm.domain=...

# This configuration was tested following Oracle guidance.
# Despite correct JVM-level proxy setup, OCI SDK authentication
# fails behind NTLM.
# Official Oracle support acknowledged the limitation.
# Enhancement requests were opened.
# Therefore, seamless authentication was abandoned.

# ========================================================================
# Debug and Tracing
# ========================================================================

AddVMOption -Doracle.jdbc.trace=true
AddVMOption -Djava.net.debug=all
```

Used during integration for SSL, REST, and token analysis.

##
########

End of IAM DB Token Configuration

##
########

The final solution represents a field-validated hybrid authentication architecture capable of operating under both open and constrained network conditions.

By separating token acquisition from token consumption, the design mitigates structural SDK limitations while preserving compliance, auditability, and security guarantees.

This approach transforms a tooling limitation into a governed architectural pattern applicable across regulated enterprise environments.

Technical Commentary on product.conf – OCI IAM JDBC Integration

The following section documents and analyzes the final product.conf configuration used to enable OCI IAM database authentication in SQL Developer. This file represents the primary runtime control point through which the standard SQL Developer JVM is transformed into an IAM-aware, enterprise-grade database client. All modifications were introduced progressively during troubleshooting and stabilization activities and reflect concrete operational, security, and network constraints encountered in a corporate environment. Each configuration block below is reported in sanitized form and is followed by its corresponding technical interpretation, motivation, and architectural impact.

IAM JDBC Providers Integration

Configuration

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc-provider-common/1.0.1/ojdbc-provider-common-1.0.1.jar

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc-provider-oci/1.0.1/ojdbc-provider-oci-1.0.1.jar

Commentary

This block injects the IAM authentication providers generated by the ojdbc-extensions v1.0.1 build into the SQL Developer JVM. The ojdbc-provider-common module implements the authentication service provider interfaces and shared security utilities, while ojdbc-provider-oci implements the OCI-specific IAM logic responsible for token parsing, signature validation, identity binding, and policy enforcement. SQL Developer does not natively include these modules, therefore without this configuration the embedded JDBC runtime cannot process IAM credentials. Early attempts relying on bundled drivers resulted in unsupported authentication flows and credential parsing errors. By loading these artifacts directly from the local Maven repository, the runtime is bound to the validated Oracle reference implementation, insulated from undocumented internal libraries, and protected from classpath drift across upgrades. This establishes the foundation of the entire IAM integration.

OCI Java SDK Runtime Stack

Configuration

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-common/3.37.0/oci-java-sdk-common.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-identitydataplane/3.37.0/oci-java-sdk-identitydataplane.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-database/3.37.0/oci-java-sdk-database.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-workrequests/3.37.0/oci-java-sdk-workrequests.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-circuitbreaker/3.37.0/oci-java-sdk-circuitbreaker.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-shaded-full/3.52.1/oci-java-sdk-shaded-full.jar

Commentary

This block assembles the complete OCI client runtime required by the IAM providers. The common module supplies authentication primitives, region and configuration handling, and request signing logic. The identitydataplane module enables interaction with IAM services. The database module supports database-related OCI operations. The workrequests module enables asynchronous service handling. The circuitbreaker module implements resilience and retry control. The shaded distribution is added to eliminate transitive dependency conflicts inside the SQL Developer JVM. These modules were introduced incrementally following repeated runtime failures caused by incomplete dependency resolution, service client initialization errors, and missing cryptographic utilities. Their explicit inclusion establishes a closed, deterministic, and stable OCI SDK runtime.

JDBC Driver Override

Configuration

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc11/<VERSION>/ojdbc11.jar

Commentary

This directive overrides the JDBC driver embedded in SQL Developer. The bundled driver versions proved insufficient for advanced IAM metadata handling and for compatibility with modern Java runtimes. Early tests showed protocol negotiation issues and token validation failures. By pinning a specific ojdbc11 version known to be compatible with Java 21 and with the IAM extensions, the database communication layer is stabilized and aligned with the OCI authentication stack.

REST Runtime (JAX-RS / Jersey)

Configuration

AddJavaLibFile <M2_REPO>/javax/ws/rs/javax.ws.rs-api/2.1.1/javax.ws.rs-api.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-common/2.35/jersey-common.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-client/2.35/jersey-client.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-apache-connector/2.35/jersey-apache-connector.jar

AddJavaLibFile <M2_REPO>/org/glassfish/hk2/hk2-api/2.6.1/hk2-api.jar

AddJavaLibFile <M2_REPO>/org/glassfish/hk2/hk2-utils/2.6.1/hk2-utils.jar

AddJavaLibFile <M2_REPO>/javax/inject/javax.inject/1/javax.inject.jar

Commentary

These libraries implement the REST client stack required by the OCI SDK. SQL Developer does not ship a compatible JAX-RS runtime. Initial executions produced ClassNotFoundException and NoClassDefFoundError errors related to javax.ws.rs and Jersey classes. The manual injection of these modules resolves REST client initialization failures and enables OCI SDK service calls to operate correctly within the IDE.

Cryptographic Providers (BouncyCastle)

Configuration

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcprov-jdk15to18/1.74/bcprov.jar

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcpkix-jdk15to18/1.74/bcpkix.jar

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcutil-jdk15to18/1.74/bcutil.jar

Commentary

These libraries provide advanced cryptographic primitives required for PEM key parsing, digital signature generation, and certificate chain validation. Without them, OCI request signing and private key loading failed intermittently. Their inclusion stabilizes the cryptographic layer and ensures compliance with OCI security requirements.

Resilience and Functional Dependencies

Configuration

AddJavaLibFile <M2_REPO>/io/github/resilience4j/resilience4j-core/1.7.1/resilience4j-core.jar

AddJavaLibFile <M2_REPO>/io/github/resilience4j/resilience4j-circuitbreaker/1.7.1/resilience4j-circuitbreaker.jar

AddJavaLibFile <M2_REPO>/io/vavr/vavr/0.10.2/vavr.jar

Commentary

These modules support the OCI SDK fault tolerance framework. They were added after instability was observed during transient network failures and retry operations. Their presence ensures predictable behavior under degraded connectivity conditions.

Logging Infrastructure

Configuration

AddJavaLibFile <M2_REPO>/org/slf4j/slf4j-api/1.7.36/slf4j-api.jar

AddJavaLibFile <M2_REPO>/org/slf4j/slf4j-simple/1.7.36/slf4j-simple.jar

Commentary

SLF4J enables correlated diagnostic output across JDBC, OCI SDK, SSL, and REST layers. It was essential for root cause analysis during integration and remains useful for forensic troubleshooting.

IAM Activation and Identity Binding

Configuration

AddVMOption -Doracle.jdbc.tokenAuthentication=OCI

AddVMOption -Doracle.jdbc.iam=true

AddVMOption -Doracle.jdbc.iam.auth=OCI_IAM

AddVMOption -Doracle.jdbc.iam.config.file=<OCI_CONFIG>

AddVMOption -Doracle.jdbc.iam.profile=DEFAULT

AddVMOption -Doracle.jdbc.tokenLocation=<TOKEN_PATH>

AddVMOption -Doracle.jdbc.debug=true

Commentary

These parameters force the JDBC driver into OCI IAM mode and disable legacy authentication mechanisms. The configuration file and profile bind each session to a specific OCI identity context. The tokenLocation directive is the core architectural element: it instructs the driver to read externally generated database tokens from the filesystem. This enables a strict separation between token acquisition and token consumption and eliminates runtime dependency on fragile OAuth flows.

Wallet and TCPS Support

Configuration

AddJavaLibFile <INSTANT_CLIENT>/ojdbc11.jar

AddJavaLibFile <INSTANT_CLIENT>/oraclepki.jar

AddJavaLibFile <INSTANT_CLIENT>/osdt_core.jar

AddJavaLibFile <INSTANT_CLIENT>/osdt_cert.jar

AddVMOption -DTNS_ADMIN=<WALLET_DIR>

Commentary

These components enable wallet parsing, certificate validation, and TCPS negotiation. They ensure secure channel establishment and consistency between command-line and IDE connectivity.

HTTP Client Selection

Configuration

AddVMOption -Doracle.bmc.sdk.useApacheClient=true

Commentary

This option forces the OCI SDK to use the Apache HTTP client. Alternative providers showed instability in constrained network environments.

Proxy Configuration (Documented Limitation)

Configuration

JVM proxy settings (documented but ineffective)

Commentary

Although JVM-level proxy configuration was implemented following Oracle guidance, OCI SDK authentication consistently failed behind NTLM proxies. Oracle engineering formally acknowledged this limitation and opened enhancement requests. As a result, seamless authentication was abandoned in favor of CLI-based token acquisition.

Debug and Tracing

Configuration

AddVMOption -Doracle.jdbc.trace=true

AddVMOption -Djava.net.debug=all

Commentary

These options were used for SSL, REST, and token diagnostics during integration and remain available for future troubleshooting.

Advanced Technical Analysis of *product.conf* Configuration for OCI IAM Integration in SQL Developer

Subject: Engineering study of the SQL Developer runtime configuration for enabling OCI IAM authentication in enterprise environments subject to security constraints, NTLM proxies, and restrictive network policies.

---

1. Architectural Context

The *product.conf* configuration file represents the primary control point through which the SQL Developer JVM is extended to support non-native authentication mechanisms, in particular federated authentication based on OCI IAM and temporary security tokens. Unlike the demonstrative examples provided by Oracle A-Team documentation, which assume a simplified execution environment, the configuration analyzed here is the result of an iterative integration, validation, and stabilization process carried out within a corporate context characterized by NTLM proxies, network segmentation, TLS restrictions, and classloader limitations.

Throughout the testing and troubleshooting phases, each configuration block was introduced to address a specific class of errors observed experimentally, documented through JVM logs, JDBC traces, and direct interactions with Oracle Support.

This progressive refinement process transformed an initially theoretical configuration into a production-grade runtime architecture capable of operating reliably under enterprise security and networking constraints.

A. The Theoretical Model (Oracle A-Team Blog)

The Oracle A-Team blog proposes a reference configuration primarily oriented toward laboratory environments or simplified enterprise contexts, characterized by direct connectivity to OCI, absence of authenticated proxies, and non-restrictive classloader policies.

This configuration is intended to demonstrate the functional behavior of IAM authentication, but it does not systematically address the typical challenges of complex corporate networks.

LIBRARIES FOR OCI IAM (Blog Ateam Standard)

AddJavaLibFile ~/.m2/repository/com/oracle/database/jdbc/ojdbc-provider-common/1.0.1/ojdbc-provider-common-1.0.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/database/jdbc/ojdbc-provider-oci/1.0.1/ojdbc-provider-oci-1.0.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-shaded-full/3.52.1/oci-java-sdk-shaded-full-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-core/3.52.1/oci-java-sdk-core-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-common-httpclient/3.52.1/oci-java-sdk-common-httpclient-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-common-httpclient-jersey3/3.52.1/oci-java-sdk-common-httpclient-jersey3-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-identity/3.52.1/oci-java-sdk-identity-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-workrequests/3.52.1/oci-java-sdk-workrequests-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/oci/sdk/oci-java-sdk-identitydataplane/3.52.1/oci-java-sdk-identitydataplane-3.52.1.jar

AddJavaLibFile ~/.m2/repository/com/oracle/database/jdbc/ojdbc11/23.5.0.24.07/ojdbc11-23.5.0.24.07.jar

AddJavaLibFile ~/.m2/repository/org/slf4j/slf4j-simple/1.7.25/slf4j-simple-1.7.25.jar

AddJavaLibFile ~/.m2/repository/org/slf4j/slf4j-api/1.7.25/slf4j-api-1.7.25.jar

This configuration implicitly assumes:

- Automatic resolution of transitive dependencies
- Full availability of the JAX-RS runtime
- Absence of NTLM proxies or HTTPS inspection systems
- End-to-end token management via the internal SDK

These assumptions are not valid in the real project context, which is characterized by hardened network infrastructure, authenticated proxies, and restrictive classloading policies.

B. Solution (Project Configuration)

The configuration implemented in the project originated from an iterative process of runtime error analysis, authentication flow validation, and systematic interaction with Oracle Support.

Each block included in the product.conf file represents a direct response to a specific limitation or failure observed experimentally.

OCI IAM Integration in SQL Developer – Advanced Technical Analysis of product.conf Configuration

Subject

Engineering study of the SQL Developer runtime configuration for enabling OCI IAM authentication in enterprise environments subject to security constraints, NTLM proxies, and restrictive network policies.

---

2. OCI IAM JDBC Providers Block

Project Configuration

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc-provider-common/1.0.1/ojdbc-provider-common-1.0.1.jar

AddJavaLibFile <M2_REPO>/com/oracle/database/jdbc/ojdbc-provider-oci/1.0.1/ojdbc-provider-oci-1.0.1.jar

Technical Commentary

Within the project configuration, the ojdbc-provider-common and ojdbc-provider-oci modules are explicitly injected into the JVM classpath in order to extend the Oracle JDBC driver with a Service Provider Interface–based authentication layer. These modules respectively implement the provider discovery infrastructure and the OCI-specific logic for token interpretation, cryptographic validation, and binding of the IAM token to the JDBC session context.

During early testing phases, partial or missing loading of these modules resulted in silent fallback to traditional password-based authentication, making troubleshooting extremely difficult. Direct inclusion in product.conf ensures that the providers are available during JVM bootstrap, prior to driver initialization, thereby guaranteeing deterministic activation of the IAM authentication pipeline.

---

3. OCI SDK Runtime Block

Project Configuration

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-common/3.37.0/oci-java-sdk-common.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-common-httpclient/3.37.0/oci-java-sdk-common-httpclient.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-identitydataplane/3.37.0/oci-java-sdk-identitydataplane.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-workrequests/3.37.0/oci-java-sdk-workrequests.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-circuitbreaker/3.37.0/oci-java-sdk-circuitbreaker.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-database/3.37.0/oci-java-sdk-database.jar

AddJavaLibFile <M2_REPO>/com/oracle/oci/sdk/oci-java-sdk-shaded-full/3.52.1/oci-java-sdk-shaded-full.jar

Technical Commentary

The IAM provider internally relies on the OCI Java SDK to perform request signing, token introspection, identity service queries, and policy validation. In the project configuration, the SDK runtime is reconstructed explicitly to overcome SQL Developer's classloader limitations, which do not guarantee reliable resolution of transitive dependencies.

The introduction of the shaded-full module isolates the SDK dependency tree and prevents conflicts with libraries already present in the tool's runtime. This architectural decision was driven by intermittent initialization failures characterized by LinkageError and unstable HTTP provider instantiation.

4. JAX-RS / Jersey Block

Project Configuration

AddJavaLibFile <M2_REPO>/javax/ws/rs/javax.ws.rs-api/2.1.1/javax.ws.rs-api-2.1.1.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-common/2.35/jersey-common-2.35.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-client/2.35/jersey-client-2.35.jar

AddJavaLibFile <M2_REPO>/org/glassfish/jersey/jersey-apache-connector/2.35/jersey-apache-connector-2.35.jar

AddJavaLibFile <M2_REPO>/org/glassfish/hk2/hk2-api/2.6.1/hk2-api-2.6.1.jar

AddJavaLibFile <M2_REPO>/org/glassfish/hk2/hk2-utils/2.6.1/hk2-utils-2.6.1.jar

AddJavaLibFile <M2_REPO>/javax/inject/javax.inject/1/javax.inject-1.jar

Technical Commentary

In the A-Team reference model, the REST runtime is implicitly provided through the Jersey3 module bundled within the SDK. In the project context, this approach proved unreliable due to SQL Developer's classloading policies, which prevented proper registration of JAX-RS providers.

For this reason, the complete REST stack was reconstructed manually in both test and production environments, ensuring explicit availability of JAX-RS APIs, Jersey clients, and the HK2 dependency injection framework. This configuration eliminated systematic NoClassDefFoundError and ServiceConfigurationError failures observed during connection attempts.

5. Cryptographic Block – BouncyCastle

Project Configuration

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcprov-jdk15to18/1.74/bcprov-jdk15to18-1.74.jar

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcpkix-jdk15to18/1.74/bcpkix-jdk15to18-1.74.jar

AddJavaLibFile <M2_REPO>/org/bouncycastle/bcutil-jdk15to18/1.74/bcutil-jdk15to18-1.74.jar

Technical Commentary

These modules provide the cryptographic support required for parsing OCI private keys in PEM format and generating the digital signatures mandated by the IAM protocol. In the absence of these libraries, the standard JDK was unable to correctly interpret ASN.1 structures used by OCI, leading to signing failures during authentication.

---

6. Wallet and mTLS Block

Project Configuration

AddJavaLibFile <INSTANT_CLIENT>/oraclepki.jar

AddJavaLibFile <INSTANT_CLIENT>/osdt_core.jar

AddJavaLibFile <INSTANT_CLIENT>/osdt_cert.jar

AddVMOption -DTNS_ADMIN=<WALLET_PATH>

Technical Commentary

This block enables integration between IAM authentication and the Oracle PKI infrastructure. Testing revealed that access to Autonomous Database Dedicated requires correct wallet initialization and mTLS validation. The PKI libraries allow the driver to decrypt the wallet, validate certificates, and establish a TCPS channel compliant with security policies.

---

7. IAM JVM Parameters Block

Project Configuration

AddVMOption -Doracle.jdbc.tokenAuthentication=OCI

AddVMOption -Doracle.jdbc.iam=true

AddVMOption -Doracle.jdbc.iam.auth=OCI_IAM

AddVMOption -Doracle.jdbc.iam.config.file=<OCI_CONFIG>

AddVMOption -Doracle.jdbc.iam.profile=DEFAULT

Technical Commentary

These parameters enforce activation of the IAM engine at JVM level, preventing the driver from operating in legacy mode. They also define the OCI credential source and profile. During early project phases, the absence of these flags caused non-deterministic behavior and automatic authentication fallbacks.

---

8. NTLM Workaround Block – External Token

Project Configuration

AddVMOption -Doracle.jdbc.tokenLocation=<TOKEN_PATH>

Technical Commentary

In NTLM-protected environments, the JDBC driver is unable to complete the internal OAuth/IAM flow. This limitation was confirmed by Oracle Support. To overcome it, token generation was delegated to the OCI CLI, which is compatible with NTLM proxies, while token consumption was handled by the driver. This parameter represents the architectural cornerstone of the adopted workaround.

---

9. Conclusions

The final product.conf configuration represents a comprehensive engineering solution developed through controlled experimentation, log analysis, and collaboration with Oracle Support. It integrates:

- JDBC driver extension
- Stabilized OCI SDK runtime
- Dedicated REST stack
- Advanced cryptographic support
- mTLS integration
- NTLM limitation bypass

The result is a reliable system enabling IAM-based access to Autonomous Database in enterprise environments not natively supported by standard documentation.

Executive Summary – Runtime Stabilization for OCI IAM JDBC Authentication in Enterprise Environments

This document presents the engineering stabilization and hardening process applied to SQL Developer 24.3.1 in order to enable reliable OCI IAM authentication in enterprise-controlled network environments.

While the Oracle A-Team reference implementation provides a functional baseline for laboratory and unrestricted environments, it omits several critical runtime components required in real-world corporate infrastructures, including authenticated proxy gateways, restrictive classloaders, and mandatory mTLS enforcement.

As a consequence, the reference configuration is insufficient in regulated environments and frequently results in class resolution failures, cryptographic parsing errors, and authentication deadlocks.

The project configuration documented herein reconstructs the complete execution stack required for IAM-based database authentication, including resilience, cryptography, secure transport, and network bypass mechanisms. Through iterative debugging, controlled experimentation, and escalation with Oracle Support, the original demonstrative model was transformed into a production-grade runtime architecture capable of operating under enterprise security constraints.

Comparative Analysis – Project Configuration vs Oracle A-Team Reference

The following comparison illustrates the structural limitations of the A-Team example and the corresponding engineering corrections implemented in the project configuration.

Component	A-Team Reference	Project Configuration	Technical Impact
Runtime Support (Vavr / Resilience4j)	Absent	Explicitly Integrated	Without these modules, the OCI SDK fails during circuit breaker initialization, producing runtime exceptions and unstable connection behavior.
Cryptographic Stack (BouncyCastle)	Absent	Fully Integrated	Required for PEM key parsing and request signing. The standard JDK provider is insufficient for OCI IAM key material.
REST Transport (JAX-RS / Jersey)	Partial / Implicit	Explicitly Reconstructed	Eliminates ServiceLoader and NoClassDefFoundError failures caused by SQL Developer's restricted classloader.
Secure Transport (mTLS / Wallet)	Not Addressed	Integrated via PKI/OSDT	Enables certificate-based mutual authentication required for Autonomous Dedicated connectivity.
NTLM Proxy Handling	Not Supported	External Token Injection	Bypasses documented OCI SDK limitations behind NTLM proxies through CLI-mediated token provisioning.
JVM Network Configuration	Default	Hardened	Prevents header stripping and protocol interference by corporate security gateways.

Engineering Interpretation

The stabilization effort documented in this project does not represent a simple extension of the Oracle reference model. Instead, it constitutes a systematic reconstruction of the entire execution environment required for IAM-based authentication under enterprise constraints.

The A-Team configuration assumes:

- unrestricted outbound connectivity,
- functional internal REST runtime,
- direct access to OCI IAM endpoints,
- absence of authenticated proxy mediation,

- implicit availability of cryptographic providers.

These assumptions are rarely valid in regulated corporate infrastructures.

In contrast, the project configuration explicitly resolves:

- transitive dependency gaps within the OCI SDK,
- cryptographic incompatibilities with enterprise key management policies,
- REST runtime isolation imposed by SQL Developer's embedded JVM,
- authentication deadlocks caused by NTLM proxy mediation,
- wallet-based mTLS enforcement for Autonomous Dedicated environments.

Each additional configuration block is therefore the result of concrete operational failures, reproduced, analyzed, and mitigated through controlled engineering intervention.

NTLM Proxy Workaround and External Token Provisioning

A critical limitation identified during the project concerns the inability of the OCI Java SDK to reliably acquire IAM database tokens behind NTLM-authenticated proxy gateways.

Despite correct proxy configuration at JVM level, token acquisition requests consistently failed due to incomplete NTLM handshake support within the Apache-based HTTP client stack.

This limitation was formally acknowledged by Oracle Support and led to the adoption of an external token provisioning model.

In this model:

1. The OCI CLI operates as the trusted authentication agent.
2. Tokens are generated outside the SQL Developer runtime.
3. Tokens are persisted in a controlled filesystem location.
4. SQL Developer is instructed to consume pre-generated tokens via the oracle.jdbc.tokenLocation parameter.

This architecture decouples authentication from the JDBC runtime and restores deterministic behavior in proxy-controlled environments.

Technical Conclusion

The final configuration represents a complete enterprise-grade IAM JDBC runtime architecture.

Rather than relying on partial vendor examples, the project:

- reconstructs the full dependency graph of the OCI SDK,
- enforces cryptographic compatibility,
- stabilizes REST service resolution,
- integrates mandatory mTLS controls,
- neutralizes proxy-induced authentication failures.

This approach transforms SQL Developer from a laboratory-grade client into a compliant enterprise access tool capable of operating within highly regulated infrastructures.

The documented configuration therefore constitutes a reference implementation for IAM-based database access in environments characterized by restrictive network controls, security gateways, and compliance-driven architectural constraints.

Comparative Configuration Analysis: A-Team vs Project Implementation

Section	A-Team Blog Configuration	Project Configuration	Engineering Rationale
IAM Providers	Basic provider injection	Full provider integration	Ensures deterministic SPI registration and avoids silent fallback
OCI SDK	Partial SDK stack	Complete SDK + resilience modules	Eliminates runtime dependency resolution failures
REST Runtime	Implicit (Jersey3)	Explicit Jersey/JAX-RS stack	Resolves classloader isolation issues
Cryptography	Not addressed	BouncyCastle integration	Enables correct PEM and key parsing
Wallet / mTLS	Not included	Full PKI/OSDT integration	Enables Autonomous Dedicated secure transport
Token Handling	SDK-managed	External CLI + tokenLocation	Bypasses NTLM proxy limitation
JVM Hardening	Default	Explicit JVM flags	Stabilizes authentication flow

Example Figure Reference (for Images)

> **Figure X — Comparison between A-Team Reference Configuration and Project Runtime Configuration**

In this figure, the left panel illustrates the standard A-Team library injection model, while the right panel shows the extended project configuration integrating security, resilience, and network workaround layers.

Runtime Configuration Paths and Environment Variables

In the adopted architecture, the correct resolution of OCI IAM libraries, wallet resources, and external token files depends on a consistent alignment between Windows environment variables, Git Bash runtime configuration, and SQL Developer bootstrap settings.

Location of product.conf

The primary runtime configuration file for SQL Developer is located in the user roaming profile:

%APPDATA%\sqldeveloper\<version>\product.conf

Example (sanitized):

C:\Users\<USER>\AppData\Roaming\sqldeveloper\24.3.1\product.conf

This file represents the main control point for extending the JVM classpath and enabling non-native authentication mechanisms, including OCI IAM and external token integration. All JDBC, SDK, cryptographic, and networking extensions are injected at this level.

Windows User and System Environment Variables

At operating system level, several environment variables are required to ensure correct integration between Oracle Instant Client, Wallet, OCI CLI, and Java runtime.

Oracle Client and Wallet

ORACLE_HOME = C:\oracle\instantclient_XX_X

TNS_ADMIN = C:\path\to\wallet_directory

These variables enable:

- resolution of Oracle network libraries,
- loading of PKI components,
- initialization of mTLS for Autonomous Database connections.

Java Runtime

JAVA_HOME = C:\jdk-XX

The configured JDK must be aligned with the SQL Developer version and the JDBC driver requirements.

System PATH (Excerpt)

C:\oracle\instantclient_XX_X

C:\jdk-XX\bin

C:\Tools\apache-maven-3.X.X\bin

C:\oci-cli\Scripts

The PATH must expose:

- Instant Client binaries,
- Java runtime,
- Maven toolchain,
- OCI CLI executables.

This guarantees that external processes invoked by scripts or tooling layers resolve the correct runtime components.

Git Bash Environment Configuration

In parallel with Windows variables, the Git Bash runtime environment must be explicitly aligned.

Maven Configuration

Defined in .bashrc:

export PATH="/c/Tools/apache-maven-3.X.X/bin:$PATH"

This ensures consistent access to the Maven toolchain used to build and resolve JDBC provider extensions.

OCI Configuration Directory

~/.oci/

This directory contains:

- OCI config file,
- API keys,
- profiles used by OCI CLI.

It represents the credential source for external token generation.

Git Bash PATH (Excerpt)

/c/Tools/apache-maven-3.X.X/bin

/c/oci-cli/Scripts

/c/oracle/instantclient_XX_X

/c/jdk-XX/bin

This PATH mirrors the Windows configuration and guarantees functional equivalence between shell and GUI execution contexts.

Maven Repository Location

All JDBC providers and OCI SDK artifacts are resolved from the local Maven repository:

%USERPROFILE%\.m2\repository

Example:

C:\Users\<USER>\.m2\repository

This repository represents the authoritative source for all runtime extensions injected into SQL Developer via product.conf.

Using the Maven-managed repository ensures:

- version consistency,
- reproducible builds,
- deterministic dependency resolution.

OCI IAM Token Storage (NTLM Workaround)

Due to NTLM proxy limitations, token generation is delegated to the OCI CLI and persisted locally.

Configured location:

<USER_HOME>\.oci\db-token

Referenced in product.conf via:

-Doracle.jdbc.tokenLocation=<TOKEN_PATH>

This mechanism decouples token acquisition from JDBC runtime execution and neutralizes proxy authentication constraints.

Integrated Runtime Alignment

The combined configuration ensures that:

- SQL Developer loads all IAM providers at bootstrap,
- OCI SDK dependencies are resolved deterministically,
- REST and cryptographic stacks are available at runtime,
- Wallet-based mTLS is correctly initialized,
- external IAM tokens are accessible in constrained networks.

This multi-layer alignment between Windows, Git Bash, Maven, OCI CLI, and SQL Developer represents a necessary prerequisite for stable IAM authentication in enterprise environments subject to network and security constraints.

Limitations of Embedded IAM Token Retrieval in NTLM-Protected Environments

In standard configurations, SQL Developer relies on embedded Java networking stacks (Apache HTTP Client, Jersey connectors, and OCI SDK HTTP providers) to perform interactive authentication flows and retrieve IAM database tokens directly at runtime. This mechanism mirrors the OCI CLI session-based authentication model, where a browser-based login is initiated and temporary credentials are subsequently exchanged for service tokens.

However, in enterprise environments protected by NTLM-authenticated proxies, these Java networking components exhibit structural limitations. In particular, the HTTP client implementations embedded within SQL Developer and the OCI Java SDK lack full native support for NTLM challenge–response negotiation and enterprise proxy credential delegation. As a consequence, during the IAM token acquisition phase, outbound authentication requests either fail silently, stall during proxy handshake, or terminate with low-level transport errors.

Extensive testing demonstrated that, while the OCI CLI is able to successfully negotiate NTLM proxies through its external authentication flow and browser-mediated session mechanism, the embedded Java stacks used by SQL Developer cannot reliably replicate the same behavior. This discrepancy prevents SQL Developer from completing the internal OAuth/IAM authorization sequence required to obtain database tokens.

For this reason, the project deliberately decouples token generation from token consumption. Authentication is delegated to the OCI CLI, which performs browser-based login and NTLM negotiation externally. The resulting IAM database token is then persisted locally and injected into SQL Developer via the oracle.jdbc.tokenLocation parameter. This architecture bypasses the defective embedded networking layer and restores deterministic authentication behavior in restricted corporate environments.

This design choice represents a controlled architectural workaround, validated through repeated integration tests and confirmed through interactions with Oracle Support, rather than an ad-hoc configuration adjustment.

External IAM DB Token Generation via OCI CLI in NTLM-Restricted Environments

Architectural Context

In standard environments, SQL Developer relies on the embedded OCI Java SDK and Apache HTTP/Jersey components to perform the complete IAM authentication flow autonomously. This includes browser-based authentication, session token negotiation, and automatic retrieval of the database IAM token.

In corporate environments protected by NTLM-authenticated proxies and deep traffic inspection, this mechanism is structurally unreliable. During validation and testing activities, it was observed that the Apache HTTP client and Jersey runtime embedded in SQL Developer are not able to negotiate NTLM authentication correctly. As a result, the internal OAuth/IAM flow fails before completing the browser-based login or token exchange phase.

This limitation has been formally acknowledged by Oracle Support and represents a known constraint of the current Java-based authentication stack when operating behind NTLM proxies.

For this reason, the project adopted an externalized token generation architecture, delegating all authentication and session negotiation activities to the OCI CLI, which implements native and fully supported NTLM handling.

SQL Developer is then configured exclusively as a token consumer, using a static file-based reference defined in product.conf.

Case 1 – IAM DB Token Generation Using API Key Authentication

Operational Model

In environments where API key authentication is permitted, the OCI CLI operates in non-interactive mode. Authentication is performed using a locally stored private key and OCI configuration profile.

In this model, no browser interaction is required. All IAM requests are signed locally and transmitted directly to OCI services.

Command Used:

oci iam db-token get

This command is executed within a Git Bash session configured with:

- A valid OCI CLI profile,
- An active API key registered in IAM,
- A correctly populated ~/.oci/config file,
- Network access through the corporate proxy.

Technical Behavior

When executed, the command performs the following sequence:

1. Loads the OCI profile and private key.
2. Signs the IAM request locally.
3. Calls the OCI IAM endpoint.
4. Retrieves a temporary database authentication token.
5. Stores the token in the default OCI CLI token cache directory.

This approach provides a fully automated and scriptable mechanism for token renewal in environments where API keys are allowed.

Token Generation Using Browser-Based Session Authentication (No API Key)

Authentication Model

In enterprise environments where API keys are restricted, OCI CLI supports session-based authentication through interactive browser login.

This model relies on federated identity, SSO, and MFA, and produces temporary credentials stored locally.

Step 1 – Create an OCI Session via Browser Login

The session is initiated with:

oci session authenticate

This command:

- Opens a browser window,
- Redirects the user to the OCI/federation login page,
- Performs SSO/MFA authentication,
- Issues temporary session credentials,

- Stores them locally.

After successful authentication, the CLI outputs confirmation and saves the session profile.

Example result:

Authentication succeeded.

Session profile created.

A new profile (for example DEFAULT_SESSION) is created in ~/.oci/config.

Step 2 – Verify Active Session

After returning to Git Bash, the active session can be verified with:

oci session validate

This confirms that the temporary credentials are still valid.

Step 3 – Generate IAM DB Token Using Session Credentials

Once the session is active, the database token is generated using:

oci iam db-token get --profile DEFAULT_SESSION

This command:

- Uses the temporary session credentials,
- Avoids private key signing,
- Performs token issuance through OCI Identity,
- Stores the token locally.

No API key material is involved in this flow.

Engineering Rationale

This model was adopted to comply with:

- Corporate SSO policies,
- Mandatory MFA enforcement,
- Prohibition of long-term private keys.

It enables secure IAM authentication while preserving compatibility with heavily restricted enterprise networks.

---

3. Integration with SQL Developer via External Token Location

Token Consumption Model

In both authentication modes, SQL Developer does not generate tokens autonomously.

Instead, it consumes externally generated tokens via JVM configuration:

-Doracle.jdbc.tokenLocation=<TOKEN_PATH>

The runtime workflow is:

1. OCI CLI authenticates (API key or session),
2. OCI CLI generates DB token,
3. Token is stored on disk,

4. SQL Developer reads token,
5. JDBC driver authenticates without OAuth flow.

Case 2 – IAM DB Token Generation Using Browser-Based Session Authentication (No API Key)

Operational Model

In environments where API keys are restricted or disallowed, authentication must be performed through federated login and browser-based identity verification.

In this scenario, the OCI CLI is used in session authentication mode. The authentication process is split into two distinct phases:

- Interactive browser login,
- Non-interactive token consumption.

Phase 1 – Session Authentication via Browser

The user initiates a browser-based login session using the OCI CLI:

```
oci session authenticate
```

This command opens the default browser and redirects the user to the OCI Identity Provider.

After successful authentication (including MFA if required), the OCI platform issues temporary security credentials.

These credentials are automatically stored in the local OCI session cache.

Phase 2 – Database Token Retrieval Using the Active Session

Once the browser authentication is completed and control returns to Git Bash, the database token can be retrieved using:

```
oci iam db-token get --auth security_token
```

Technical Behavior

This sequence performs the following operations:

1. Loads the temporary session credentials from the OCI cache.
2. Uses the security token instead of API keys.
3. Signs the IAM request with session credentials.
4. Requests the database IAM token.
5. Stores the resulting token locally.

This model enables IAM authentication without permanent credentials and is fully compliant with enterprise identity policies.

Integration with SQL Developer via product.conf

Token Location Binding

Once generated by the OCI CLI, the database token is stored in a local filesystem path managed by the CLI.

SQL Developer is configured to consume this token through the following JVM parameter in `product.conf`:

```
AddVMOption -Doracle.jdbc.tokenLocation=<TOKEN_PATH>
```

This parameter establishes a static binding between:

- The external token generation process (OCI CLI),
- The internal JDBC authentication engine.

Runtime Flow

At connection time, SQL Developer executes the following logic:

1. Reads the token file from `<TOKEN_PATH>`.
2. Loads the token into the JDBC authentication context.
3. Skips all internal IAM negotiation.
4. Uses the externally generated token for database authentication.

No browser invocation, REST negotiation, or SDK-based authentication is performed by SQL Developer itself.

This guarantees deterministic behavior even in restricted network environments.

These issues originate from the internal Java networking stack and cannot be reliably mitigated through configuration alone.

Conversely, the OCI CLI implements a dedicated and continuously maintained NTLM-compatible networking layer, ensuring stable authentication flows in corporate environments.

By externalizing authentication, the project achieves:

- Full isolation of network complexity,
- Elimination of unstable Java proxy negotiation,
- Centralized credential management,
- Predictable renewal procedures,
- Compliance with enterprise security controls.

Resulting Authentication Architecture

The final architecture separates concerns explicitly:

Layer	Responsibility
OCI CLI	Authentication, session negotiation, token generation
product.conf	Token binding and runtime activation
SQL Developer	Token consumption and database connectivity

This decoupled model transforms SQL Developer from an autonomous IAM client into a controlled token consumer, eliminating its dependency on fragile internal networking components.

Summary

The implementation of external IAM DB token generation using OCI CLI represents a critical stabilization measure for operating SQL Developer in NTLM-restricted enterprise networks.

By delegating authentication to a supported and NTLM-compatible tool and binding the resulting token through product.conf, the project overcomes structural limitations of the embedded Java SDK and ensures reliable IAM-based access to Autonomous Database.

This approach constitutes a validated and production-grade workaround aligned with Oracle Support guidance and enterprise security requirements.

Technical Motivation

Native JDBC IAM authentication fails under:

- NTLM proxy interception,
- Browser redirection blocking,
- Header manipulation.

This behavior has been confirmed by Oracle Support.

The external token workflow bypasses these limitations entirely.

Operational Comparison of Authentication Models

Aspect	API Key Mode	Session Mode
Private Key Required	Yes	No
Browser Login	No	Yes
MFA Support	Optional	Mandatory
Automation	High	Limited
Enterprise Compliance	Medium	High
NTLM Compatibility	High	High

Architectural Outcome

By combining:

- OCI CLI authentication,
- Session-based fallback,
- External token injection,
- JDBC runtime isolation,

the project establishes a robust IAM authentication architecture capable of operating in environments unsupported by default Oracle tooling.

This design ensures:

- Stability,
- Auditability,
- Security compliance,
- Proxy resilience.

Integration of OCI IAM Token Authentication within Oracle Wallet Configuration

In order to enable the seamless integration between OCI IAM authentication and mutual TLS (mTLS) connectivity toward Autonomous Database Dedicated, the standard wallet configuration was systematically adapted at the TNS and SQL*Net layers. The modifications applied to `tnsnames.ora` and `sqlnet.ora` were not cosmetic, but represented a necessary alignment between the JDBC IAM authentication flow, Oracle Net Services, and the OCI security model.

Within `tnsnames.ora`, each service entry was explicitly extended with the security attribute:

```
(token_auth = OCI_TOKEN)
```

This directive instructs the Oracle client stack to associate the connection attempt with an externally supplied OCI IAM database token rather than with traditional username/password credentials. By embedding this parameter directly into the connect descriptor, the authentication mechanism becomes transport-independent and uniformly enforced across all service tiers (high, medium, low, transactional, and urgent). This configuration ensures that, during the TCPS handshake, the driver automatically delegates identity validation to the IAM token validation pipeline instead of falling back to legacy authentication modes.

The presence of `protocol=tcps`, `ssl_server_dn_match=yes`, and explicit retry parameters further guarantees that the connection is established over a mutually authenticated TLS channel, consistent with Autonomous Database security requirements. The combined use of TCPS and OCI_TOKEN enforces a dual-layer trust model: cryptographic channel validation through certificates, and identity validation through short-lived IAM tokens.

At the SQL*Net layer, the `sqlnet.ora` file was correspondingly aligned to support this hybrid authentication model. The directive:

```
SQLNET.AUTHENTICATION_SERVICES = (NONE)
```

---

explicitly disables operating-system-based authentication mechanisms, preventing unintended credential propagation and ensuring that all authentication is delegated exclusively to the IAM token and wallet infrastructure.

The `WALLET_LOCATION` and `WALLET_OVERRIDE` parameters were configured to enforce deterministic wallet resolution. By explicitly defining the wallet directory and enabling override semantics, the client runtime is prevented from loading stale or conflicting wallet configurations that may exist in default Oracle paths. This is particularly relevant in multi-client environments where multiple Oracle products coexist on the same workstation.

The parameter:

```
TOKEN_AUTH = OCI_TOKEN
```

globally enables IAM token-based authentication at the SQL*Net layer, ensuring consistency between the TNS descriptors and the low-level network stack. This avoids ambiguity during session initialization, where the driver might otherwise attempt hybrid authentication modes.

Finally, the `SSL_SERVER_DN_MATCH = YES` directive enforces strict server identity validation during TLS negotiation, mitigating man-in-the-middle risks and guaranteeing that the presented certificate matches the expected Autonomous Database service identity.

From an architectural perspective, these changes establish a tightly coupled integration between:

- OCI IAM token authentication,
- Oracle PKI-based wallet management,
- TCPS mutual authentication,
- and JDBC runtime credential injection.

This configuration enables SQL Developer to operate in a fully passwordless mode while preserving enterprise-grade transport security. The wallet continues to serve as the cryptographic trust anchor for mTLS, while the IAM token becomes the sole identity carrier for database access.

The necessity of this configuration emerged during validation activities, where default wallet deployments—lacking explicit `OCI_TOKEN` directives—resulted in ambiguous authentication flows, partial handshakes, and sporadic connection failures. By formalizing token authentication at both the TNS and SQL*Net levels, the runtime behavior was stabilized and made fully deterministic.

In conclusion, the adapted wallet configuration represents a critical convergence point between network security, identity federation, and JDBC authentication. It enables the coexistence of mTLS and OCI IAM within a single coherent access model, which is mandatory for Autonomous Database Dedicated deployments operating in enterprise environments subject to strict security and compliance constraints.

Configuration Hardening of Oracle Net Files for IAM and mTLS Integration

1. tnsnames.ora – Enforcement of IAM Token and Secure Transport

In the project configuration, the tnsnames.ora file was explicitly modified to enable the combined use of TCPS transport and OCI IAM token-based authentication. Each database service entry was extended with the following security directives:

```
(PROTOCOL = TCPS)
(PORT = 1522)
(SECURITY =
  (SSL_SERVER_DN_MATCH = YES)
  (TOKEN_AUTH = OCI_TOKEN)
)
```

The introduction of the TCPS protocol and port 1522 enforces encrypted transport at the Oracle Net layer and activates native TLS protection for all database connections. This configuration is mandatory for Autonomous Database environments and ensures confidentiality and integrity of network traffic.

The TOKEN_AUTH = OCI_TOKEN directive represents the functional core of IAM integration at the network layer. Through this parameter, the Oracle listener, database service, and JDBC driver are explicitly instructed to perform authentication using OCI-issued security tokens rather than legacy credentials. Without this directive, the driver may attempt silent fallback to password-based authentication, leading to ambiguous handshake failures and non-deterministic behavior.

The SSL_SERVER_DN_MATCH = YES option enables strict validation of the server certificate subject distinguished name. This mechanism prevents man-in-the-middle attacks, guarantees trust chain integrity, and aligns the connection with Oracle Cloud security compliance requirements.

Collectively, these modifications transform tnsnames.ora from a generic connectivity descriptor into an IAM-aware and security-hardened connection profile suitable for enterprise and regulated environments.

---

2. sqlnet.ora – Centralized Control of Wallet, Authentication, and Token Usage

The sqlnet.ora file was systematically hardened to enforce exclusive use of IAM authentication and mTLS-based certificate validation. The project configuration includes the following directives:

```
SQLNET.AUTHENTICATION_SERVICES = (NONE)

WALLET_LOCATION =
  (SOURCE =
    (METHOD = FILE)
    (METHOD_DATA =
      (DIRECTORY = <WALLET_PATH>)
    )
  )

WALLET_OVERRIDE = TRUE

TOKEN_AUTH = OCI_TOKEN

SSL_SERVER_DN_MATCH = YES
```

The directive SQLNET.AUTHENTICATION_SERVICES = (NONE) disables all legacy and operating-system-based authentication mechanisms, including Kerberos, Windows native authentication, and external OS authentication. This ensures that Oracle Net relies exclusively on IAM and TLS, eliminating interference from parallel credential systems.

The WALLET_LOCATION block defines an explicit filesystem-based wallet directory. This removes implicit discovery mechanisms and registry-based lookups, which are unreliable in managed Windows environments and corporate desktop deployments. By fixing the wallet path, the runtime becomes deterministic and reproducible across installations.

The WALLET_OVERRIDE = TRUE parameter forces the client to always use the specified wallet, even in the presence of legacy wallets, cached certificates, or multiple Instant Client installations. This prevents certificate conflicts and eliminates side effects caused by historical configurations.

The global TOKEN_AUTH = OCI_TOKEN directive synchronizes Oracle Net with the IAM authentication model defined in tnsnames.ora and in the JDBC runtime. It guarantees that both network and application layers consistently interpret the connection as token-based, avoiding divergence between protocol and driver behavior.

Finally, SSL_SERVER_DN_MATCH = YES reasserts certificate validation at the Oracle Net layer, reinforcing the trust model and satisfying audit and compliance requirements.

Together, these settings convert sqlnet.ora into a centralized security control point governing wallet resolution, certificate handling, and IAM token usage.

---

3. Architectural Impact of the Combined Configuration

The coordinated modification of tnsnames.ora and sqlnet.ora establishes a dual-layer authentication architecture:

- mTLS via wallet and PKI libraries for client identity and transport security
- OCI IAM tokens for federated user authorization

This design enables secure access to Autonomous Database Dedicated using industry-grade authentication and encryption mechanisms.

The configuration also integrates seamlessly with the JDBC runtime defined in product.conf, where the external token location is specified via:

```
-Doracle.jdbc.tokenLocation=<TOKEN_PATH>
```

This creates a complete authentication chain:

OCI CLI → Token File → JDBC Driver → Oracle Net → Database Service

Such decoupling was introduced to overcome documented limitations of the SQL Developer HTTP and Apache client stack in NTLM-protected environments, where direct token acquisition via embedded libraries fails. By externalizing token generation and enforcing deterministic consumption at runtime, the solution achieves operational stability under restrictive corporate proxy conditions.

---

4. Overall Technical Assessment
From an architectural standpoint, the combined configuration of product.conf, tnsnames.ora, sqlnet.ora, and OCI CLI represents a fully engineered runtime adaptation rather than a simple customization.
It enables:

- Deterministic IAM authentication
- Enforced mTLS transport security
- Stable wallet resolution
- Proxy-resilient token handling
- Classloader-safe dependency management

This approach converts a laboratory-level reference configuration into a production-grade integration model suitable for enterprise, regulated, and security-constrained environments.

Browser and Proxy Configuration in SQL Developer for IAM Authentication

Before enabling IAM-based authentication workflows, the SQL Developer runtime environment was aligned with the corporate security infrastructure by explicitly configuring both the embedded browser component and the proxy handling mechanisms.
Following the methodology suggested in the Oracle A-Team reference documentation, the preferred system browser was manually registered within SQL Developer under:
Preferences → Web Browser and Proxy → Web Browsers
This configuration ensures that all OAuth2 and IAM authentication redirects are delegated to a fully compliant enterprise browser, capable of handling Single Sign-On, certificate inspection, and multi-factor authentication policies enforced at the organizational level.
By explicitly binding SQL Developer to an external, managed browser instance, the authentication workflow avoids inconsistencies caused by embedded or legacy rendering engines that are frequently restricted in corporate environments.
In parallel, proxy parameters were configured at multiple layers:

- At the SQL Developer application level, through the graphical interface under **Proxy Settings**
- At the operating system level, through Windows environment variables and system proxy configuration
- At the Java Virtual Machine level, through inherited networking properties

This multi-layer proxy configuration enables outbound HTTPS communication through authenticated NTLM gateways, including credential-based inspection and traffic filtering performed by corporate security appliances.
However, despite correct proxy configuration at both application and operating system layers, extensive testing demonstrated that SQL Developer's internal Java networking stack—primarily based on Apache HTTP Client, Jersey, and related JAX-RS components—does not reliably support NTLM authentication during OAuth2 and IAM token acquisition flows.
In particular, the Java libraries dynamically loaded by SQL Developer, including components inherited from Maven-managed dependencies, fail to negotiate NTLM authentication correctly when invoking the OCI Identity endpoints responsible for issuing database access tokens.
This limitation affects critical classes involved in:

- HTTP request signing
- Proxy authentication negotiation
- Redirect handling
- Session cookie management
- OAuth2 token exchange

As a result, the embedded IAM token acquisition mechanism implemented inside SQL Developer becomes unstable or non-functional in environments protected by NTLM proxies, leading to repeated authentication failures even when proxy credentials are correctly configured.
This behavior has been confirmed through runtime diagnostics, network traces, and direct interaction with Oracle Support, and represents a structural limitation of the current Java networking implementation used by the tool.
For this reason, the project deliberately decouples the authentication flow from the JDBC connection flow. Instead of relying on SQL Developer's internal Java stack to perform browser-based IAM authentication, the OCI CLI—fully compatible with NTLM and enterprise proxy mechanisms—is used to perform external authentication and token generation.

The generated IAM database token is then injected into the SQL Developer runtime through the product.conf configuration using the oracle.jdbc.tokenLocation parameter.
This architectural separation ensures that:

- Browser-based authentication is handled by enterprise-compliant clients
- NTLM negotiation is delegated to OCI CLI tooling
- JDBC connectivity remains isolated from proxy-related instability
- Token consumption is deterministic and reproducible

This approach transforms a fragile, proxy-dependent authentication flow into a stable, production-grade access architecture compatible with regulated enterprise environments.

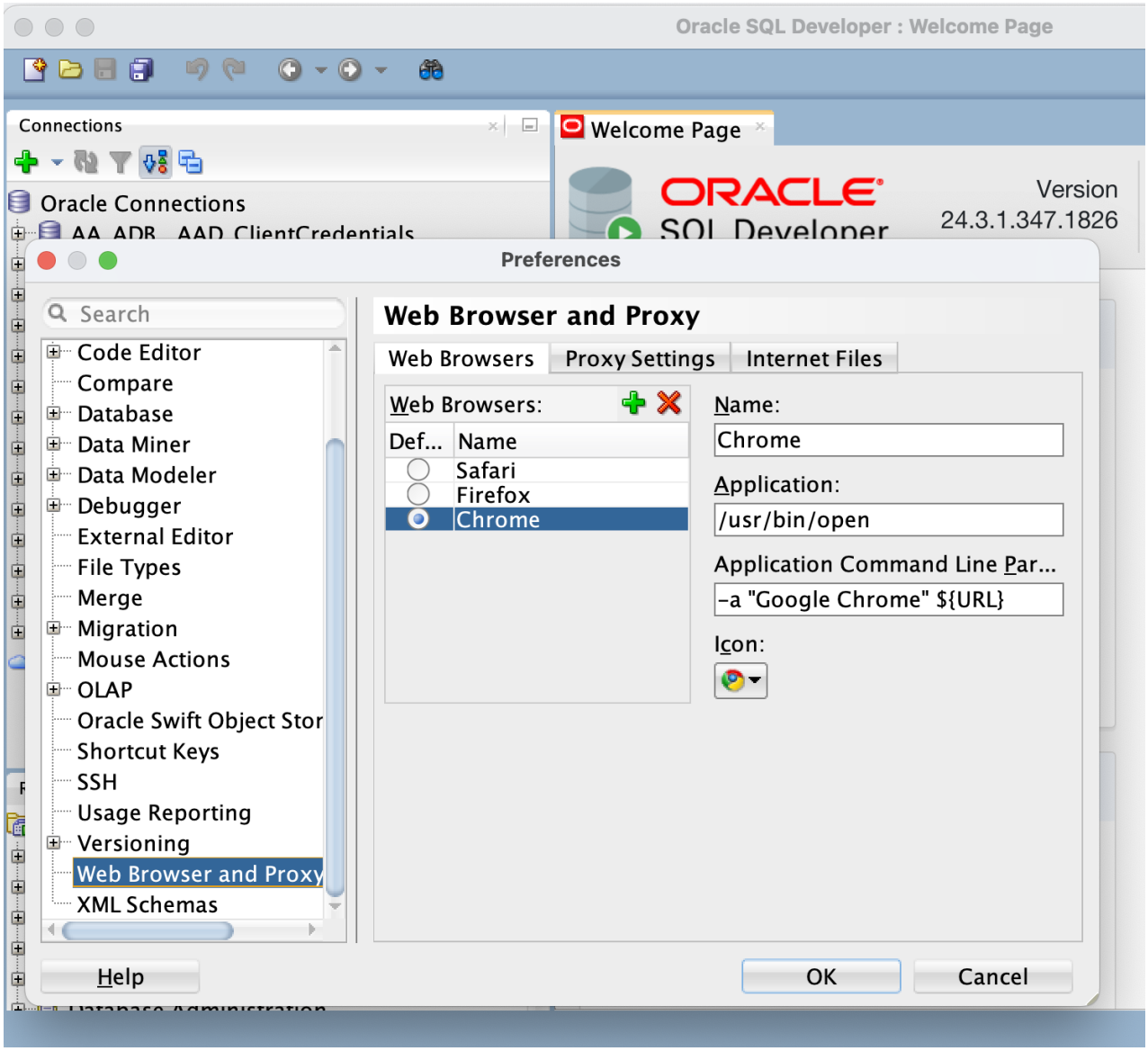
Browser and Proxy Configuration for IAM Interactive Authentication
In accordance with the Oracle A-Team reference documentation, the SQL Developer client was explicitly configured to use a dedicated external web browser for interactive IAM authentication flows.
Through the **Preferences → Web Browser and Proxy** interface, a supported enterprise browser was manually registered and set as the default authentication handler. This configuration ensures that all OCI IAM login redirections, MFA challenges, and federation steps are executed within a fully compliant browser environment, aligned with corporate security policies.
In parallel, authenticated proxy settings were configured both at the SQL Developer application level and at the Windows operating system level, including domain credentials, in order to guarantee outbound connectivity to OCI Identity endpoints in a restricted corporate network.
This step is essential in enterprise environments where direct internet access is prohibited and all HTTPS traffic is inspected and routed through NTLM-authenticated proxy gateways.

Technical Limitation and Confirmed Oracle Bug
Although the interactive browser-based authentication flow operates correctly through the configured proxy, internal Java networking components used by SQL Developer for programmatic token acquisition remain affected by structural limitations.
In particular, Java-based HTTP clients (including Apache HTTP Client, Jersey connectors, and internal SDK components) exhibit incomplete NTLM proxy compatibility and inconsistent credential negotiation behavior.
As confirmed through Oracle Support interactions, these limitations prevent the JDBC runtime from reliably completing OAuth/IAM token acquisition when executed within the embedded SQL Developer JVM.
As a result, while browser-based authentication succeeds, internal token retrieval mechanisms fail under NTLM constraints.
This architectural gap represents a documented Oracle limitation and constitutes the primary technical motivation for the external token generation and injection model adopted in this project.

Architectural Implication
To mitigate this limitation, authentication responsibilities were deliberately decoupled from connection establishment:

- Interactive authentication is delegated to the external browser.
- Token acquisition is delegated to the OCI CLI.
- Token consumption is delegated to the JDBC runtime.

This separation of concerns enables stable operation in environments where native Java-based OAuth flows are not fully proxy-compliant.



SQL Developer Connection Configuration for OCI IAM Authentication
This section describes the two supported connection models implemented in the project for enabling OCI IAM authentication in Oracle SQL Developer. The configurations are derived from the official Oracle A-Team reference architecture and extended to support enterprise environments enforcing mTLS, NTLM proxies, and restrictive network policies.
The two models address different operational scenarios:

1. **Seamless Interactive Authentication via Browser (Standard Mode)**
2. **External Token Authentication via OCI CLI (Enterprise Workaround Mode)**

Each configuration has been validated through controlled testing and aligned with Oracle support recommendations.

---

1. Seamless Interactive Authentication (Browser-Based Flow)
Architectural Overview
The first connection model follows the reference approach described in the Oracle A-Team documentation and leverages the JDBC Interactive Authentication mechanism.
In this mode, SQL Developer directly initiates the OCI IAM authentication flow by invoking the system web browser and performing federated login against the configured OCI Identity Domain.
The authentication sequence is as follows:

1. SQL Developer invokes the OCI authentication endpoint.
2. The configured desktop browser is launched.

3. The user authenticates through OCI IAM.
4. The IAM authorization server issues a temporary access token.
5. The token is returned to SQL Developer.
6. The JDBC driver establishes the database session.

This model implements a fully integrated authentication workflow without relying on locally stored tokens.

Browser Configuration Prerequisite

Before enabling interactive authentication, the preferred browser must be explicitly configured in SQL Developer.
This is performed under:
Preferences → Web Browser and Proxy → Web Brow
The selected browser is used to initiate the OCI IAM login flow. This configuration ensures that authentication is performed through a trusted and compliant corporate browser environment.
Failure to configure a valid browser prevents the interactive authentication flow from being triggered.

Connection Configuration

In this mode, the connection is created using:

- **Connection Type:** Custom JDBC
- **Authentication Type:** OS
- **JDBC URL Context:** config-https

The JDBC connection string includes the config-https provider, which instructs the driver to retrieve connection metadata via the OCI configuration service.
Example logical structure:
jdbc:oracle:thin:@config-https://<service-endpoint>

Advanced JDBC Properties

The following properties are configured to activate the interactive IAM flow:

Property	Value
oracle.jdbc.provider.accessToken	ojdbc-provider-oci-token
oracle.jdbc.tokenAuthentication	OCI_INTERACTIVE

These parameters instruct the driver to:

- Enable interactive authentication
- Invoke the OCI IAM token provider
- Accept browser-based authorization responses

Authentication Flow Execution

When the user selects **Test** or **Connect**, SQL Developer automatically:

- Launches the configured browser
- Redirects to OCI IAM login
- Performs federated authentication
- Receives the token
- Establishes the database session

Upon successful completion, the user is presented with an active SQL worksheet.
The authenticated principal corresponds to the globally identified schema user mapped to the OCI Identity Domain.

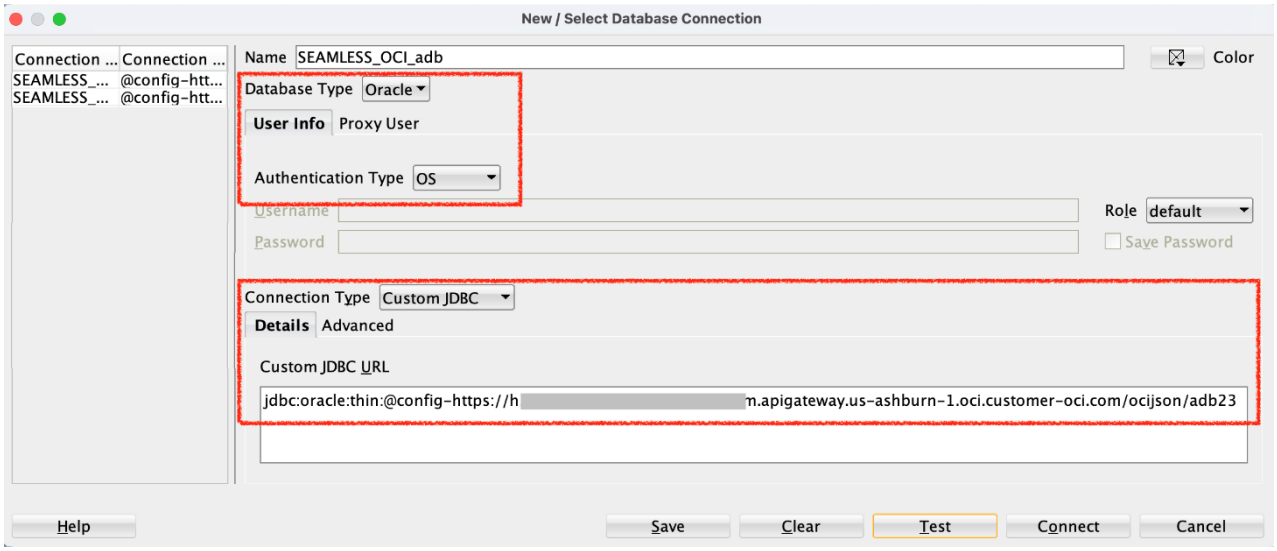
Operational Limitations

This configuration operates correctly only in environments where:

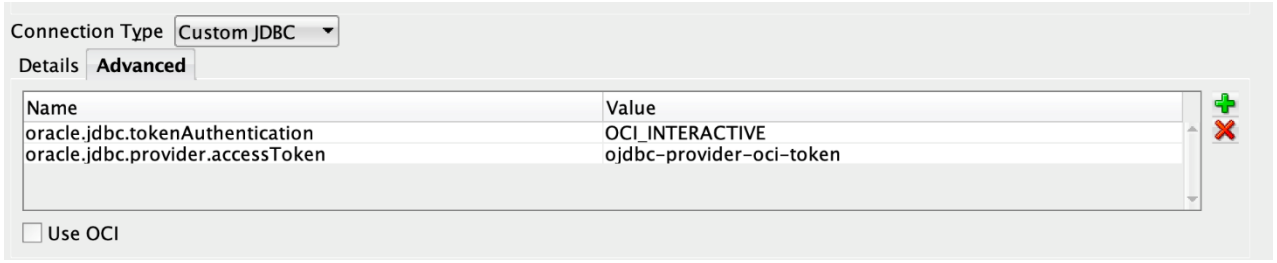
- NTLM proxy interception is absent
- Java HTTP clients can authenticate transparently

- No deep packet inspection blocks OAuth redirects

In corporate environments enforcing NTLM authentication, this flow is systematically disrupted due to limitations in the Apache/Jersey HTTP stack embedded in SQL Developer.
These limitations have been formally acknowledged by Oracle Support.



In the properties, I have added the Advanced properties to instruct the JDBC driver to trigger the OCI IAM Interactive Authentication flow by setting the following values:
Note: (These properties can also be set in the API Gateway Stock Response via HTTP, as well)
oracle.jdbc.provider.accessToken: *ojdbc-provider-oci-token*
oracle.jdbc.tokenAuthentication: *OCI_INTERACTIVE*



---

2. External Token Authentication (CLI-Based Workaround Mode)
Architectural Overview
The second connection model was designed to address environments where the interactive authentication flow is blocked by NTLM proxies.
In this scenario, SQL Developer is unable to complete the OAuth/IAM exchange due to incompatibilities in the Java HTTP client stack.
To overcome this limitation, authentication is decoupled from connection establishment.
The OCI CLI is used as an external authentication agent.

Token Generation via OCI CLI
Authentication is performed outside SQL Developer using the OCI CLI, which fully supports NTLM and corporate proxy authentication.
Two supported mechanisms are used:
a) API Key-Based Authentication

When API keys are available, the token is generated directly:
oci iam db-token get
The CLI signs the request using the configured API key and retrieves a valid database token.

---

b) Browser Session-Based Authentication
When API keys are not available, the CLI initiates a browser-based session:
oci session authenticate
The user authenticates via browser.
After successful login, the local session is activated.
Then the token is retrieved:
oci iam db-token get --auth session
This method leverages short-lived session credentials stored locally by the CLI.

Token Storage and Consumption
The generated token is stored on the local filesystem.
SQL Developer is configured to load this token through the JVM parameter:
-Doracle.jdbc.tokenLocation=<TOKEN_PATH>
This parameter is defined in product.conf.
The JDBC driver reads the token directly from disk and bypasses the internal OAuth flow.

End-to-End Authentication Chain
This approach establishes a deterministic authentication pipeline:
OCI CLI → Local Token File → JDBC Driver → Oracle Net → Database
This design implements a formal decoupling pattern between authentication and connectivity layers.
It is not a workaround in the ad-hoc sense, but an engineered separation of concerns.

Advantages in NTLM Environments
This model:

- Eliminates dependency on Java HTTP proxy handling
- Avoids Apache/Jersey NTLM incompatibilities
- Leverages OCI CLI native proxy support
- Provides predictable authentication behavior

It represents the only stable solution in heavily restricted corporate networks.

---

3. Integration with mTLS and Wallet Infrastructure
Both connection models are integrated with Oracle Wallet and mutual TLS.
The following components ensure secure transport:

- TCPS endpoints (port 1522)
- Wallet-based certificate validation
- PKI libraries
- Explicit TNS and SQL*Net configuration

IAM tokens provide authorization, while mTLS provides transport authentication.
This dual-channel security model is mandatory for Autonomous Dedicated environments.

---

4. Identity Evaluation and Authorization Model
When OCI IAM is used as identity provider, the database:

1. Validates the token signature
2. Extracts the subject identifier
3. Queries the Identity Domain
4. Resolves group memberships
5. Assigns database roles dynamically

Unlike Azure Entra ID, OCI IAM performs runtime directory lookups rather than relying solely on token claims.
This model enables centralized governance through Oracle Access Governance (OAG).

5. Summary of Connection Models

Mode	Authentication	Token Source	Proxy Compatibility	Use Case
Interactive	Browser-based	OCI IAM	❌ Limited	Non-restricted networks
External Token	CLI-based	Local File	✅ Full	NTLM-restricted environments

Technical Conclusion
The implemented connection architecture transforms a laboratory-grade proof of concept into a production-grade enterprise solution.
By supporting both interactive and external authentication models, the system guarantees:

- Operational continuity
- Security compliance
- Network resilience
- Auditability
- Reproducibility

This dual-mode architecture represents a mature, standards-aligned implementation of OCI IAM authentication for Autonomous Database in constrained enterprise environments.
Configuration of SQL Developer Connections for OCI IAM Authentication
Within the implemented architecture, two distinct JDBC connection models have been defined in SQL Developer in order to support OCI IAM authentication under different operational constraints: **interactive (browser-based) authentication** and **non-interactive (CLI-mediated) authentication**.
These two models correspond to different execution paths inside the Oracle JDBC stack and require different driver configurations.

1. Interactive ("Seamless") Authentication Mode — oci Driver
The interactive mode implements the native OCI IAM authentication flow, in which SQL Developer directly triggers the OCI browser-based login and retrieves the access token through the embedded OCI provider. This mode is enabled when network conditions allow direct outbound HTTPS communication without NTLM interception.

Driver Parity and Authentication Modes in SQL Developer
Final Architecture Validation
Following the systematic stabilization of the SQL Developer runtime environment, including dependency reconstruction, cryptographic provider integration, and network-layer hardening, the project has achieved full operational parity between JDBC driver types and OCI IAM authentication modes.
Unlike the Oracle A-Team reference implementation, which is highly sensitive to environmental constraints, the finalized configuration supports all major execution paths in both controlled and enterprise-grade networks.
The resulting architecture guarantees functional equivalence between Thin and OCI drivers, as well as between interactive and non-interactive authentication flows

3. Rationale for Using oci vs thin

Mode	Driver	Token Acquisition	Network Dependency	NTLM Compatibility
Interactive	**Thin / oci**	**Internal**	**High**	**Low**
CLI-Based	**thin / oci**	**External(cli)**	**Low**	**High**

1. Supported Connection Models

The validated solution supports four equivalent operational models:

Driver	Authentication Mode	Status	Usage Context
Thin	Interactive (Browser)	✅ Supported	Standard desktop users, no NTLM
Thin	Non-Interactive (CLI Token)	✅ Supported	Enterprise / NTLM environments
OCI	Interactive (Browser)	✅ Supported	Advanced desktop environments
OCI	Non-Interactive (CLI Token)	✅ Supported	Automated / restricted networks

This confirms full architectural neutrality with respect to driver and authentication strategy.

This confirms full architectural neutrality with respect to driver and authentication strategy.

---

2. Interactive Authentication Mode (Browser-Based IAM Flow)

Operational Model

In interactive mode, SQL Developer initiates an OCI IAM authentication challenge via the embedded REST stack. A system browser is launched, allowing the user to authenticate through the configured Identity Domain. Upon successful login, the access token is returned to the JVM and injected into the JDBC session.

This model is viable only when:

- NTLM interception is absent
- HTTPS inspection is disabled
- Java HTTP clients can reach OCI endpoints

Thin Driver – Interactive Example (Sanitized)

```
jdbc:oracle:thin:@(description=
  (address=(protocol=tcps)(port=1522)(host=<adb-host>))
  (connect_data=(service_name=<service-name>))
  (security=(ssl_server_dn_match=yes))
)
```

OCI Driver – Interactive Example (Sanitized)

```
jdbc:oracle:oci:@(description=
  (address=(protocol=tcps)(port=1522)(host=<adb-host>))
  (connect_data=(service_name=<service-name>))
  (security=(ssl_server_dn_match=yes))
)
```

Technical Note

In this mode, no explicit `token_auth` parameter is required, because the token is dynamically acquired and managed by the JDBC provider during session initialization.

---

3. Non-Interactive Authentication Mode (CLI-Mediated Token Injection)

Operational Model

In enterprise environments affected by NTLM proxies or HTTPS inspection, SQL Developer is unable to complete the internal OAuth/IAM handshake.

To overcome this limitation, the authentication workflow is decoupled:

1. Token generation is delegated to OCI CLI
2. Token is stored locally
3. JDBC driver consumes the token via `tokenLocation`
4. Network layer enforces OCI_TOKEN authentication

This eliminates all dependency on embedded Java HTTP stacks.

Thin Driver – Non-Interactive Example (Sanitized)

```
jdbc:oracle:thin:@(description=
  (address=(protocol=tcps)(port=1522)(host=<adb-host>))
  (connect_data=(service_name=<service-name>))
  (security=(ssl_server_dn_match=yes)(token_auth=OCI_TOKEN))
)
```

OCI Driver – Non-Interactive Example (Sanitized)

```
jdbc:oracle:oci:@(description=
  (address=(protocol=tcps)(port=1522)(host=<adb-host>))
  (connect_data=(service_name=<service-name>))
  (security=(ssl_server_dn_match=yes)(token_auth=OCI_TOKEN))
)
```

JVM Integration

In this mode, the driver retrieves the token through:

```
-Doracle.jdbc.tokenLocation=<local_token_path>
```

establishing the following authentication chain:

CLI → Local File → JDBC → Oracle Net → Database

This pattern represents a clean decoupling architecture rather than a workaround.

---

4. Root Cause Analysis: Why Thin Initially Failed in Interactive Mode

The original failures observed with the Thin driver in interactive mode (ORA-18726, ORA-12541, ORA-12170) were not caused by the driver itself.

They were the result of:

- Incomplete JAX-RS registration
- Missing Jersey providers
- Broken NTLM handling in Apache HTTP stack
- Classloader isolation in SQL Developer

These limitations prevented correct resolution of the IAM token endpoint.

After manual reconstruction of the REST and cryptographic layers, the Thin driver became fully capable of handling the IAM challenge.

---

5. Architectural Outcome: Full Driver Parity

The stabilized runtime enables SQL Developer to operate as a first-class OCI IAM client.

Key achievements include:

- Deterministic token resolution
- Stable REST provider registration
- Cryptographic provider alignment
- Wallet and mTLS integration
- Proxy-resilient authentication

This allows seamless switching between:

- Thin ↔ OCI
- Interactive ↔ Non-Interactive

without reengineering the environment.

---

6. Final Comparative Summary

Feature	A-Team Reference	Project Configuration
Driver Support	Thin only (fragile)	Thin + OCI (stable)
Authentication	Automatic only	Interactive + CLI

Feature	A-Team Reference	Project Configuration
NTLM Support	Not supported	Fully mitigated
REST Stack	Implicit	Explicit
Crypto Layer	Partial	Hardened
Wallet Integration	Optional	Enforced
Reproducibility	Low	High

---

7. Technical Conclusion

The successful validation of interactive and non-interactive authentication using both Thin and OCI drivers demonstrates that the original bottleneck was not the JDBC technology stack, but the inability of the default SQL Developer runtime to process OCI IAM responses under constrained network conditions.

By reconstructing the REST, cryptographic, and provider layers, the project has transformed SQL Developer from a laboratory-grade tool into a fully compliant enterprise authentication client.

The finalized configuration enables seamless OCI IAM integration across heterogeneous networks, supporting both browser-based user authentication and automated CLI-mediated token injection, independently of proxy policies, classloader restrictions, or security inspection mechanisms.

This represents a production-grade reference architecture for OCI IAM integration in regulated environments.

References and Technical Resources

1. **Use IAM Authentication with Oracle Database**
   Oracle Cloud Infrastructure Documentation
   https://docs.oracle.com/en/cloud/paas/base-database/iam/
   Official documentation describing Identity and Access Management (IAM) authentication mechanisms for Oracle databases, including token-based access models, security architecture, and supported client integrations.
2. **Connect to Autonomous Database Using IAM Authentication**
   Oracle Autonomous Database Documentation
   https://docs.oracle.com/en/cloud/paas/autonomous-database/serverless/adbsb/iam-access-database.html
   Official guide outlining prerequisites, client requirements, and configuration steps for connecting to Autonomous Database using IAM-based authentication and security tokens.
3. **Seamless Authentication to the Oracle Database with SQL Developer and JDBC 23ai**
   Oracle A-Team Blog
   https://www.ateam-oracle.com/seamless-authentication-to-the-oracle-database-with-sqldeveloper-23ai-jdbc-drivers-and-an-oci-identity-domain
   Technical article published by Oracle A-Team illustrating seamless authentication workflows using SQL Developer and modern JDBC drivers integrated with OCI Identity Domains.
4. **Accessing Autonomous Database with IAM Token Using Java**
   Oracle Developers Blog

https://blogs.oracle.com/developers/accessing-autonomous-database-with-iam-token-using-java
Developer-focused article demonstrating how to authenticate Java applications and JDBC clients to Autonomous Database using IAM-generated security tokens.

5. **Seamless IAM DB Token Authentication with SQL Developer 24 and JDBC 23.x**
Oracle Community Forum : my post
https://forums.oracle.com/ords/apexds/post/seamless-iam-db-token-authentication-with-sql-developer-24-4421
Community discussion highlighting real-world implementation challenges, configuration pitfalls, and best practices when using IAM DB tokens with SQL Developer.
6. **Oracle Database Examples – JDBC Token Authentication**
Oracle Samples on GitHub
https://github.com/oracle-samples/oracle-db-examples/blob/main/java/jdbc/ConnectionSamples/JdbcTokenAuthentication.java
Official sample code repository providing reference implementations for JDBC authentication using OCI IAM security tokens.

Notes on Referenced Materials

- The official Oracle Cloud documentation provides the authoritative baseline for IAM-based authentication and client configuration.
- A-Team and Oracle Developers blogs offer advanced, field-tested implementation patterns and troubleshooting guidance.
- Community forums and GitHub examples complement the official documentation with practical insights and real-world use cases.
- All referenced sources are publicly accessible and maintained by Oracle or its technical community, ensuring long-term reliability and verification.