10

MLE Security

MLE utilizes a number of methods to support good security practices. This includes enforcing runtime state isolation, system and object privileges, and providing monitoring options.

Topics

System and Object Privileges Required for Working with JavaScript in MLE

Depending on the project's requirements, different privileges can be granted to users and or roles, allowing them to interact with JavaScript in the database.

Administrators should review application requirements carefully and only grant the minimum number of privileges necessary to users. This is especially true for system privileges, which are very powerful and should only be granted to trusted users.

The minimum privilege required to work with MLE JavaScript code is the right to execute JavaScript code in the database. MLE distinguishes between dynamic MLE execution based on `DBMS_MLE` and MLE execution using MLE modules and environments.

Creating stored code in JavaScript requires additional privileges to create JavaScript schema objects in your own schema.

The most powerful privileges available in MLE allow super-users to create, alter, and drop MLE schema objects in any schema, not just their own. As with all privileges in Oracle Database, those with `ANY` in their name are most powerful and should only be granted to trusted users if deemed absolutely necessary.

> **Note:**
>
> Object privileges on modules and environments do not grant access to an application, for example, the combination of source code and user context defined by a call specification (or through `DBMS_MLE`). This is achieved by granting access to the procedure or function object of the call specification.

> **See Also:**
>
> - Necessary Privileges for Creating MLE Modules and Environments in ANY Schema for more about handling system privileges
> - *Oracle Database Security Guide* for more information about privileges in the Oracle Database

Topics

Necessary Privileges for the Execution of JavaScript Code

Before you can execute any JavaScript code in your own schema, the following object grant must have been issued to your user account:

```
GRANT EXECUTE ON JAVASCRIPT TO <role | user>
```

The `EXECUTE ON JAVASCRIPT` privilege does not include dynamic execution of JavaScript using `DBMS_MLE`. If you wish to make use of `DBMS_MLE`, an additional privilege is required:

```
GRANT EXECUTE DYNAMIC MLE TO <role | user>
```

Necessary Privileges for Using the NoSQL API

In cases where MLE JavaScript code references the Simple Oracle Document Access (SODA), the `SODA_APP` role must be granted to the user or role:

```
GRANT SODA_APP <role | user>
```

Necessary Privileges for Creating MLE Schema Objects

If you wish to create MLE modules and environments in your own schema, further system privileges are required:

```
GRANT CREATE MLE TO <role | user>
```

In case any MLE module is to be exposed to the database's SQL and PL/SQL layers in the form of call specifications, you also require the right to create PL/SQL procedures:

```
GRANT CREATE PROCEDURE TO <role | user>
```

It is highly likely that you will require further system privileges, depending on your use case, to create additional schema objects such as tables, indexes, and sequences. Beginning with Oracle Database 23ai, the `DB_DEVELOPER_ROLE` role allows administrators to grant the necessary privileges to developers in their local development databases quickly. The role can be granted as shown in the following snippet:

```
GRANT DB_DEVELOPER_ROLE TO <role | user>
```

> **See Also:**
>
> *Oracle Database Security Guide* for more information about the `DB_DEVELOPER_ROLE` role

Necessary Privileges for Creating MLE Modules and Environments in ANY Schema

Additional privileges can be granted to power users and administrators, allowing them to create, alter, and drop MLE schema objects in *any* schema.

```
GRANT CREATE ANY MLE TO <role | user>
GRANT DROP ANY MLE TO <role | user>
GRANT ALTER ANY MLE TO <role | user>
```

As with all privileges in Oracle Databases featuring `ANY` in their name, these are very powerful and should only be granted after a thorough investigation to trusted users. For this reason, only the `DBA` role and the `SYS` account have been granted these privileges. The use of these system privileges is audited by the `ORA_SECURECONFIG` audit policy.

To create MLE call specifications in schemas other than your own requires the right to `CREATE ANY PROCEDURE` to be granted as well:

```
GRANT CREATE ANY PROCEDURE TO <role | user>
```

Just like the previously listed system privileges, `CREATE ANY PROCEDURE` is audited by the same audit policy, `ORA_SECURECONFIG`.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about the `ORA_SECURECONFIG` audit policy

Necessary Privileges for Post-Execution Debugging

It is possible to allow other database users to collect debug information for MLE modules they don't own. By default, MLE owners can use post-execution debugging on their own MLE modules without specific grants. It is possible to grant the ability to collect debug information to a different role or user, allowing them to use post-execution debugging of JavaScript code on your behalf as the module owner:

```
GRANT COLLECT DEBUG INFO ON <module> TO <role | user>
```

> **Note:**
>
> You can elect to grant the execute privilege on MLE module calls created as PL/SQL code with definer's rights to users in other schemas. In this case, there is no need to grant other users any additional privileges.

> **Note:**
>
> Object privileges on modules and environments do not grant access to an application, for example, the combination of source code and user context defined by a call specification (or through `DBMS_MLE`). This is achieved by granting access to the procedure or function object of the call specification.

> **See Also:**
>
> Post-Execution Debugging of MLE JavaScript Modules for more information on post-execution debugging

Security Considerations for MLE

Besides the use of account privileges, MLE employs several other methods to ensure a high level of security.

Topics

- MLE_PROG_LANGUAGES Initialization Parameter

MLE_PROG_LANGUAGES Initialization Parameter

A new initialization parameter, `MLE_PROG_LANGUAGES`, allows administrators to enable and disable Multilingual Engine completely or selectively enable certain languages. It takes the values `ALL`, `JAVASCRIPT`, or `OFF` and it can be set at multiple levels:

- Container Database (CDB)
- Pluggable Database (PDB)
- Database session

If the parameter is set to `OFF` at CDB level, it cannot be enabled at PDB or session level. The same logic applies for PDB and session level: if MLE is disabled at the PDB level, it cannot be enabled at session level.

> **Note:**
>
> In Oracle Database 23ai, MLE supports JavaScript as its sole language. Setting the parameter to `ALL` or `JAVASCRIPT` has the same effect.

> **Note:**
>
> Setting `MLE_PROG_LANGUAGES` to `OFF` prevents the execution of JavaScript code in the database, it does not prevent the creation or modification of existing code.

> **See Also:**
>
> *Oracle Database Reference* for more information about `MLE_PROG_LANGUAGES`

Execution Contexts

When executing JavaScript code in the database, MLE uses execution contexts to isolate runtime state such as global variables and other important information. Execution contexts are created implicitly when using modules and environments and explicitly when using `DBMS_MLE`.

Regardless of the choice of JavaScript invocation, execution contexts are designed to prevent information leak.

The scope of JavaScript state never exceeds the lifetime of a database session. As soon as the session ends, either gracefully or forcefully, session state is discarded. If state needs to be

preserved between sessions, you must persist it by storing it in a schema. If needed, state can be discarded by calling `DBMS_SESSION.reset_package()`.

As an additional security measure, you can optionally specify the use of a restricted execution context, which disallows access to the database state. The `PURE` keyword is used in the creation of environments and in inline call specifications to indicate the use of a restricted context. An environment created using `PURE` can be referenced in module call specifications and using `DBMS_MLE`. PURE execution serves as a method to isolate certain code, such as third-party JavaScript libraries, from the database itself. This isolation can reduce the attack surface of supply chain attacks, in which access to the database state is a security concern.

See Also:

- About Restricted Execution Contexts for more information about the `PURE` keyword and restricted contexts
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_SESSION`

Runtime State Isolation

An MLE call specification is a PL/SQL unit referencing a function in an MLE module with an optional MLE environment attached. When you invoke a call specification in a session, the corresponding MLE module is loaded, the optional environment is applied, and the function specified in the call specification's signature clause is executed.

Before execution can begin, a corresponding execution context must be created (implicitly). Whether a new execution context is created or an existing context is reused depends on multiple factors, specifically:

- The MLE module referenced in the call specification
- The corresponding MLE environment
- The database user executing the call specification

Separate execution contexts are created to prevent information leak as well as undesired side effects such as global variables in a module being overwritten by accident.

With each invocation of a call specification, additional execution contexts are created. This is done so that modules cannot interfere with one another.

The main criteria for creating execution contexts in a user session are the MLE module name and the corresponding MLE environment. Call specifications referring to different combinations of MLE module and environment lead to different individual execution contexts being created.

Further separation between execution contexts is performed based on the user invoking the call specification.

Example 10-1 Runtime State Isolation Scenario

This example provides a sample scenario for runtime state isolation. Database user `USER1` creates the following MLE schema objects:

```
CREATE OR REPLACE MLE MODULE isolationMod LANGUAGE JAVASCRIPT AS
```

```
let id;      // global variable

export function doALotOfWork() {
  // a dummy function simulating a lot of work
  // the focus is on modifying a global variable

  id = 10;
}

export function getId() {

  return (id === undefined ? -1 : id)
}
/

CREATE OR REPLACE MLE ENV isolationEnv;

CREATE OR REPLACE PACKAGE context_isolation_package AS

  -- initialise runtime state
  procedure doALotOfWork as
    mle module isolationMod
    signature 'doALotOfWork()';

  -- access a global variable (part of session state)
  function getId return number as
    mle module isolationMod
    signature 'getId()';

  -- same function signature as before but referencing an environment
  function getIdwEnv return number as
    mle module isolationMod
    env isolationEnv
    signature 'getId()';
END;
/
```

When `USER1`, the owner of the MLE module, environment, and call specification (package), calls `context_isolation_package.doALotOfWork()`, the global variable (`id`) is initialized to `10`.

```
BEGIN
    context_isolation_package.doALotOfWork();
END;
/
```

Because `context_isolation_package.getId()` references the same MLE module and the same (default) environment as `context_isolation_package.doALotOfWork()`, the user's session has access to the global variable:

```
SELECT CONTEXT_ISOLATION_PACKAGE.getId;

     GETID
----------
        10
```

When the combination of user, MLE module, and environment change, a new execution context is created. Although `context_isolation_package.getIdwEnv()` references the same MLE module as `getID()` and the user doesn't change, the function cannot retrieve the value of the global variable from the previously created execution context:

```
SELECT CONTEXT_ISOLATION_PACKAGE.getIdwEnv;

 GETIDWENV
----------
        -1
```

A value of `-1` indicates that the global variable in the JavaScript module was found to be uninitialized.

If `USER1`, as the owner of the MLE call specification, grants the execute privilege on the package to another user, let's say `USER2`, a different execution context is created for `USER2` even though the same function is called:

```
GRANT EXECUTE ON CONTEXT_ISOLATION_PACKAGE TO user2;
```

When `USER2` tries to read the value of the `ID`, a new context is created and the return value indicating an uninitialized context is returned:

```
SELECT user1.CONTEXT_ISOLATION_PACKAGE.getid;

     GETID
----------
        -1
```

In this example, module and environment are identical between `USER1` and `USER2` as per the call specification. However, the fact that the function is called by a different user causes a new execution context to be created.

Database Security Model

The fewer privileges granted to program units, accounts, and roles, the less likely it is for them to be misused. As with every application, the principle of granting only the minimum number of necessary privileges should be followed. This is especially true in higher-tier environments like production. Technologies such as Privilege Analysis can be used to track down unnecessary privileges, allowing you to revoke them after careful regression testing.

Each MLE call specification is created within its own security context. The context includes information such as:

- The value of the `AUTHID` clause (definer or invoker)
- Whether or not privileges are inherited in invoker's rights calls
- Code Based Access Control
- Current user
- The qualified schema name
- Enabled Roles and Privileges in the absence of code based access control (CBAC) and invoker's rights

The combination of these attributes forms the security context of a code unit such as a MLE call specification or module. Note that no such security context exists for the JavaScript code stored in an MLE module.

PL/SQL allows you to easily change these attributes for each PL/SQL unit. A procedure can be executed with the invoker's rights or the definer's rights, roles can be attached to PL/SQL units, and cross-schema (execute) grants are commonplace. With each execution of a PL/SQL unit the security context may potentially change. This applies equally to MLE call specifications.

The situation is different with JavaScript code: the security context does not change for JavaScript-to-JavaScript calls. JavaScript functions do not have any notion of associated invoker's or definer's rights, or roles granted on the function itself. All of these apply only to (PL/SQL) call specifications.

JavaScript executed using `DBMS_MLE` is a little more strict when it comes to its security context. The combination of currently active user, roles/privileges, and schema in effect are recorded at the time the execution context is created by calling `DBMS_MLE.create_context()`. This combination must not change until the JavaScript code is executed and the context is removed, or else an error is thrown.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about Privilege Analysis

Considerations for Using MLE Call Specifications and Modules from Different Schemas

The same consideration that is used for other database applications written in, for example, PL/SQL apply for MLE JavaScript code as well. If a user is granted access to execute code from a schema other than their own, care needs to be taken to ensure the extent to which the code can use privileges of the calling user is appropriate.

Unlike PL/SQL, MLE JavaScript code stored in an MLE module is not associated with a particular set of roles, or any other notion of determining the security context in which the JavaScript code executes. From a high-level view, there are two important cases for cross-schema use of privileges:

1. `USER1` invokes a call specification located in `USER2`'s schema. The `AUTHID` clause of the call specification in `USER2`'s schema determines whether the code owned by `USER2`'s schema executes with the privileges of the invoker (`USER1`) or definer (`USER2`). In case of an invoker's rights call specification, potentially attached roles (CBAC) and the setting of `INHERIT PRIVILEGES` determine the active roles and privileges in addition to those granted by `USER1` by roles or direct grants.
2. `USER1` creates a call specification `CallSpec_A` for a module `Module_A` owned by `USER1`. `CallSpec_A` imports a JavaScript module `Module_B` owned by a different schema, `USER2`. The JavaScript code in `Module_B` is imported into an execution context created for `USER1`'s call specification `CallSpec_A`. The JavaScript code in `Module_B` executes with the same privileges as any other JavaScript code in this execution such as in `Module_A`. `USER1` must ensure that the code in `Module_B` is trustworthy and appropriate to execute with these privileges.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about roles in definer's rights and invoker's rights PL/SQL units

Auditing MLE Operations in Oracle Database

Auditing is the monitoring and recording of configured database actions. As with any other auditable operations in Oracle Database, the use of MLE-related system privileges can be recorded.

Oracle provides the `ORA_SECURECONFIG` audit policy with the database. Starting with Oracle Database 23ai, the audit policy includes the use of the following MLE system privileges:

- `CREATE ANY MLE`
- `ALTER ANY MLE`
- `DROP ANY MLE`

Administrators and security teams need to create and enable additional security policies if auditing the creation of MLE schema objects, including MLE modules, environments, and call specifications, is desired.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about auditing in Oracle Database

JavaScript Security Best Practices

Details concerning the best practices when using features of MLE with JavaScript are described.

Topics

Using Bind Variables for Security and Performance

The MLE JavaScript SQL driver allows you to use string concatenation to build SQL commands, including the predicates used in queries and DML statements. It is strongly recommended to avoid this bad practice as it is a major source for SQL injection attacks. Not

only is the use of bind variables in SQL statements more secure than string concatenation but it is also more efficient as it allows the database to reuse the cursor in the shared pool.

If it is not possible to avoid the creation of dynamic SQL, ensure that you validate input to your code and scan for malicious content. The built-in `DBMS_ASSERT` package provides a wealth of functions designed to mitigate against SQL injection attacks. It does not offer complete protection but its use is very much recommended as it allows you to verify the following:

- The input string is a qualified SQL name
- The input string is an existing schema name
- The input string is a simple SQL name
- The input parameter string is a qualified SQL identifier of an existing SQL object

The use of bind variables for better security and scalability is not limited to a single programming language such as JavaScript, it equally applies to every development project using Oracle Database.

> **See Also:**
>
> - Server-Side JavaScript API Documentation for information about using bind variables with `mle-js-oracledb`
> - *Oracle Database Development Guide* for more details regarding bind variables and their impact on performance and security

Example 10-2 Using Bind Variables Rather than String Concatenation

In this example, the `SELECT` statement accepts a bind variable rather than concatenation the input variable, `managerID`, to the SQL command.

```
CREATE OR REPLACE MLE MODULE select_bind LANGUAGE JAVASCRIPT AS

import oracledb from "mle-js-oracledb";

export function numEmployeesByManagerID(managerID) {

  const conn = oracledb.defaultConnection(managerID);
  const result = conn.execute(
    `SELECT count(*) FROM employees WHERE manager_id = :1`,
    [ managerID ]
  );

  return result.rows[0][0];
}
/
```

Example 10-3 Use DBMS_ASSERT to Verify Valid Input

In this example, the function `createTempTable()` creates a private temporary table to hold intermediate results from a batch process. The function takes a single argument: the name of

the temporary table to be created (minus the prefix). The function checks if the parameter passed to it is a valid SQL name.

```
CREATE OR REPLACE MLE MODULE dbms_assert_module LANGUAGE JAVASCRIPT AS

import oracledb from "mle-js-oracledb";

export function createTempTable(tableName) {
  const conn = oracledb.defaultConnection();
  let result;
  let validTableName;

  try {
    result = conn.execute(
      `SELECT dbms_assert.qualified_sql_name(:tableName)`,
      [tableName]
    );
    validTableName = result.rows[0][0];
  } catch (err) {
    throw (`'${tableName}' is not a valid table name`);
    return;
  }

  result = conn.execute(
    `CREATE PRIVATE TEMPORARY TABLE ora\$ptt_${validTableName} (id number)`
  );
}
/
```

If the table name passed to the function passes the test, it is then used to create a private temporary table using the default `private_temp_table_prefix`.

Generic Database and PL/SQL Specific Security Considerations

Because all JavaScript code is accessed eventually via a PL/SQL call specification, it is important to understand the implications of using PL/SQL as well. The following concepts are of particular importance:

- The difference between invoker's rights and definer's rights
- Code Based Access Control (CBAC)
- The impact of `INHERIT PRIVILEGES` in invoker's rights code
- Role grants and direct grants, both object as well as system privileges

You should always aim to only require the minimum security privileges (object and system) for JavaScript code to execute. This is especially important when you consider the use of external third-party JavaScript code.

Administrators should consider the use of encryption for both data at rest as well as data in motion.

> **See Also:**
>
> - *Oracle Database Security Guide* for more information about generic database-related security aspects
> - *Oracle Database Transparent Data Encryption Guide* for information about encrypting data at rest using Transparent Data Encryption (TDE)

Supply Chain Security

Access to the rich community ecosystem is one of the advantages of using JavaScript in Oracle Database. Rather than creating functionality in-house and potentially duplicating effort, existing JavaScript can be used instead. While this is a convenient method for developing applications, it comes with certain risks.

In past years, the term supply chain attach has been used to describe the fact that certain popular open-source JavaScript modules have been abandoned by the original maintainers. Bad actors have taken some of these projects, becoming maintainers but only to inject malicious code into the source. The next time a project references such a compromised module, they incorporate the malicious code.

The same principles applied to client-side development apply to server-side development with MLE. Developers and security teams must be aware that code in the application executes with potentially elevated privileges. These can be abused by malicious code to compromise confidentiality, integrity, and availability properties of the application. For that reason, extra care must be taken to ensure third-party code is trustworthy and that the minimum number of privileges is granted to it. Many companies have a dedicated security team for vetting open-source modules prior to granting their approval to use them. At the very least, you should audit the JavaScript code that you are about to include in your project and document the result.

It is possible to lock a given version of an open-source module using a mechanism like the `package-lock.json` file so as not to get caught out if a new version of a module is distributed. Automatically pulling the latest version of an external code dependency is bad practice and should always be avoided.

In the case of JavaScript in MLE, JavaScript code executes with the database privileges that are in effect for the associated execution context. JavaScript code can retrieve and modify data stored in the database according to these privileges. Malicious code can leverage these privileges to modify the database in an inappropriate manner.

As a consequence, be sure to grant the privileges to create MLE modules carefully and only grant these in environments where they are essential. If possible, avoid granting the [`CREATE` | `ALTER` | `MODIFY`] `ANY` system privileges at all.

You should also review the `INHERIT PRIVILEGES` settings in the context of invoker's rights procedures. Once the settings for `INHERIT PRIVILEGES` are reviewed and secured according to industry best practice, consider the use of invoker's rights for MLE call specifications.

Additional higher levels of security for invoker's rights procedures can be achieved by implementing code based access control (CBAC). Using CBAC, developers can associate roles to PL/SQL units without having to elevate the privileges of the schema or invoker.

> **See Also:**
>
> *Oracle Database Security Guide* for details about the `INHERIT PRIVILEGES` privilege

Software Bill of Material

Every project relying on external code in projects is strongly encouraged to maintain a record of all software components (including versions) that are bundled in a deployed application artifact.

The software bill of material (SBOM) is the key tool to use when reacting swiftly to a newly published vulnerability is of utmost importance. Exploits are almost guaranteed to be used immediately after a vulnerability has been published. Knowing exactly which version of a third-party library is in use allows you to save crucial time in preparing a response.

In addition to storing the actual code, MLE modules feature a metadata field that can be used to store arbitrary metadata with the module. In particular, it can be used to store an SBOM that describes all JavaScript libraries bundled in the module. The field is not interpreted by the MLE runtime. Content and format are entirely up to you.

> **See Also:**
>
> MLE JavaScript Modules and Environments for more information about creating MLE modules and providing metadata to them

Using the Database to Store State

Applications written using MLE JavaScript code should not deviate from established patterns such as storing application state in tables. This allows you to make the best use of the rich number of security features available for Oracle Database.

In particular, you should not rely on JavaScript state that exceeds the boundaries of one stored procedure or function call.

Oracle Database has great support for JSON, offering both a relational as well as a NoSQL API. The database's JSON API is a natural candidate for MLE JavaScript code to store state. Storing state in Oracle Database provides a better programming model than application state, especially when it come to data persistence and transactional consistency.

> **See Also:**
>
> *Oracle Database JSON Developer's Guide* for information about using JSON with Oracle Database

Example 10-4 Using Bind Variables Rather than String Concatenation

In this example, the `SELECT` statement accepts a bind variable rather than concatenation the input variable, `managerID`, to the SQL command.

```
CREATE OR REPLACE MLE MODULE select_bind LANGUAGE JAVASCRIPT AS

import oracledb from "mle-js-oracledb";

export function numEmployeesByManagerID(managerID) {

  const conn = oracledb.defaultConnection(managerID);
  const result = conn.execute(
    `SELECT COUNT(*) FROM employees WHERE manager_id = :1`,
    [ managerID ]
  );

  return result.rows[0][0];
}
/
```

Example 10-5 Use DBMS_ASSERT to Verify Valid Input

In this example, the function `createTempTable()` creates a private temporary table to hold intermediate results from a batch process. The function takes a single argument: the name of the temporary table to be created (minus the prefix). The function checks if the parameter passed to it is a valid SQL name.

```
CREATE OR REPLACE MLE MODULE dbms_assert_module LANGUAGE JAVASCRIPT AS

import oracledb from "mle-js-oracledb";

export function createTempTable(tableName) {
  const conn = oracledb.defaultConnection();
  let result;
  let validTableName;

  try {
    result = conn.execute(
      `SELECT dbms_assert.qualified_sql_name(:tableName)`,
      [tableName]
    );
    validTableName = result.rows[0][0];
  } catch (err) {
    throw (`'${tableName}' is not a valid table name`);
    return;
  }

  result = conn.execute(
    `CREATE PRIVATE TEMPORARY TABLE ora\$ptt_${validTableName} (id number)`
  );
}
/
```

If the table name passed to the function passes the test, it is then used to create a private temporary table using the default `private_temp_table_prefix`.

Disabling Multilingual Runtime

In the case where a security vulnerability is detected in JavaScript code, you can prevent JavaScript code from execution by disabling the JavaScript runtime. Setting the initialization parameter `MLE_PROG_LANGUAGES` to `OFF` does not stop the database from accepting new code (such behavior prevents the implementation of a code fix) but it does stop anyone from executing JavaScript code.

Applications should be written with that option in mind. Once the MLE runtime is disabled, an error is thrown. Rather than showing the raw error to the end user, a more accessible error message should be created.

Although JavaScript does not have a specific lockdown feature, using the `MLE_PROG_LANGUAGES` parameter allows you to disable the MLE runtime at the session, PDB (lockdown profiles operate at this level), or CDB level. The `COMMON_SCHEMA_ACCESS` feature bundle in the lockdown profile can be used to disable MLE DDL.

MLE Security Examples

Example scenarios are used to demonstrate security features used by MLE. The examples use a varying degree of separation between MLE modules, environments, and the necessary grants to enable the utilized functionality.

Note that the examples are not fully usable on their own. The actual JavaScript code is not as important as the application's structure, such as:

- The schemas in which the code is located
- The call specification's syntax
- The roles and privileges granted

Topics

Business Logic Stored in MLE Modules

In this scenario, a user provides functionality implemented in JavaScript that is bound to a particular schema and relies on being executed as a particular user with certain privileges.

This scenario covers the typical case of a back-end application centered around a single schema containing all necessary tables, indices, etc. Most importantly, the business logic is implemented as stored code in the database.

The JavaScript implementation in the form of MLE modules and an MLE environment is encapsulated in a single schema. Access to the functionality is only exposed using MLE call specifications based on one or multiple modules. Users of the application are granted execute privileges on (PL/SQL) call specifications only. No further privileges on MLE modules and environment are granted, nor are they necessary.

Consequently, the owner of the MLE modules controls access to the application through the `AUTHID` clause attached to the MLE call specifications. The pseudo-code in Example 10-6 demonstrates this scenario.

Example 10-6 Business Logic Stored in MLE Modules

In this example, the application schema is referred to as `APP_OWNER`. Note how MLE modules and environments are restricted to the `APP_OWNER` schema.

```
-- MLE Module containing helper functions commonly used by the application
CREATE MLE MODULE app_owner.helper_module LANGUAGE JAVASCRIPT AS

export function setDebugLevel(level) {
  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- An MLE Environment allowing other MLE Modules to import the helper module
CREATE MLE ENV app_owner.helper_module_env IMPORTS (
  'helperModule' module helper_module
);

-- The main application module imports the helper module for common tasks
CREATE MLE MODULE app_owner.orders_module LANGUAGE JAVASCRIPT AS

import { setDebugLevel } from "helperModule";

export function newOrder() {

setDebugLevel("INFO");
  // ... JavaScript code ...
}

export function delivery() {
  setDebugLevel("WARN");
  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- The call specification is all the end users need to be granted
-- access to. The execute privilege to this definer's rights procedure
-- (created and executed with the app_owner's database privileges)
-- is all that needs granting to the application role.

CREATE app_owner.package orders_pkg AS

  PROCEDURE new_order AUTHID DEFINER AS
```

```
    MLE MODULE orders_module
    ENV helper_module_env
    SIGNATURE 'newOrder()';

  PROCEDURE delivery AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV helper_module_env
    SIGNATURE 'delivery()';

END order_pkg;
/

GRANT EXECUTE ON app_owner.package orders_pkg TO app_role;
```

Generic Data Processing Libraries

In this scenario, generic JavaScript functionality is logically grouped inside a database schema. The JavaScript code is neither functionally nor logically tied to any existing database objects. In other words, the processing logic is stateless.

As there is no relation to any database schema objects such as tables or views, object grants are of no concern. The JavaScript code purely transforms functional arguments. Examples for such libraries include machine learning code, image manipulation like scaling, cropping, changes of resolution, etc. Other use cases include input validation or JSON processing.

The main purpose of the MLE modules deployed in such a fashion is to provide you with a common set of JavaScript tools that can be used in your own applications. Therefore, there aren't any pre-defined MLE call specifications provided. Instead, the schema containing these modules grants the execute privilege on MLE modules. It is up to the grantee to define MLE call specifications matching the use case. If necessary, MLE environments can be created alongside the MLE modules with respective grants to developers wishing to use the functionality created. Example 10-7 illustrates this scenario.

Example 10-7 Generic Data Processing Libraries

```
-- Common functionality potentially referenced by multiple applications
-- is grouped in a database schema. This particular MLE Module provides
-- input validation
CREATE MLE MODULE library_owner.input_validator_module
  LANGUAGE JAVASCRIPT USING BFILE(js_src_dir, 'input_validator.js');
/

-- Another MLE module provides common machine learning functionality
CREATE MLE MODULE library_owner.commom_ml_module
  LANGUAGE JAVASCRIPT USING BFILE(js_src_dir, 'commom_ml_lib.js');
/

-- Rather than a Call Specification as demonstrated in Example 10-6,
-- this time the MLE Modules themselves are exported for use
-- in a different schema: frontend_app
GRANT EXECUTE ON library_owner.input_validator_module TO frontend_app;
GRANT EXECUTE ON library_owner.commom_ml_module TO frontend_app;

-- frontend_app makes explicit use of a select few functions exported
-- by the MLE modules
CREATE PACKAGE input_validator_pkg AS
```

```
  FUNCTION checkEMail(p_email VARCHAR2) RETURN BOOLEAN AS
    MLE MODULE library_owner.input_validator_module
    SIGNATURE 'checkEmail(string)';

  FUNCTION checkZIPCode(p_zipcode VARCHAR2) RETURN BOOLEAN AS
    MLE MODULE library_owner.input_validator_module
    SIGNATURE 'checkZIPCode(string)';

  -- additional functionality ...
END;
/
```

The grouping of common, stateless JavaScript code is not limited to a single schema. Further separation by feature, functionality, or maintainer is possible as well.

Generic Libraries in Business Logic

This scenario utilizes business logic contained in a single schema and extends functionality using generic libraries.

This example extends the scenarios demonstrated by Example 10-6 and Example 10-7. It is conceivable that the domain-specific business logic might require extension by common functionality such as logging or debugging. The latter can be written generically so that other applications can include it as well. There are numerous advantages to that approach including, but not limited to a unified framework for auxiliary functions.

In Example 10-8, the business logic in the `APP_OWNER`'s schema, defined in Example 10-6, is extended with the previously introduced validation and machine learning functionality from Example 10-7.

There is no "best way" to work with MLE modules and environments in the database. It always depends on your particular use case. The included examples simply provide some background on how application logic can be grouped or separated, depending on a project's needs.

Example 10-8 Use Generic Libraries in Business Logic

```
-- Centrally managed JavaScript code library in the LIBRARY_OWNER schema
CREATE MLE MODULE library_owner.commom_ml_module
  LANGUAGE JAVASCRIPT USING BFILE(js_src_dir, 'commom_ml_lib.js');
/

-- The grant makes the module available to APP_OWNER
GRANT EXECUTE ON library_owner.commom_ml_module TO app_owner;

-- Business logic in schema APP_OWNER makes use of the common ML library
CREATE MLE MODULE app_owner.helper_module LANGUAGE JAVASCRIPT AS

export function setDebugLevel(level) {
  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- A generic MLE environment references both APP_OWNER's as well as
```

```
-- LIBRARY_OWNER's MLE modules
CREATE MLE ENV app_owner.all_dependencies_env imports (
  'helperModule' module helper_module
  'commonML'     module library_owner.commom_ml_module
);

-- The main application module imports the helper module for common tasks
-- as well as the common machine learning module provided by LIBRARY_OWNER
CREATE MLE MODULE app_owner.orders_module LANGUAGE JAVASCRIPT AS

import { setDebugLevel } from "helperModule";
import { churnRate }     from "commonML";

export function newOrder() {

  setDebugLevel("INFO");
  // ... JavaScript code ...
}

export function delivery() {
  setDebugLevel("WARN");
  // ... JavaScript code ...
}

export function estimateChurnRate() {

  // This function was imported from the common ML library
  // (an MLE module not stored in APP_OWNERs schema)
  const cr = churnRate();

  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- the call specification is all the end-users need to be granted
-- access to. The execute privilege to this definer rights procedure
-- (created and executed with the app_owner's database privileges)
-- is all that needs granting to the application role.

CREATE app_owner.package orders_pkg AS

  PROCEDURE new_order AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV all_dependencies_env
    SIGNATURE 'newOrder()';

  PROCEDURE delivery AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV all_dependencies_env
    SIGNATURE 'delivery()';

  FUNCTION estimateChurnRate AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV all_dependencies_env
```

```
    SIGNATURE 'estimateChurnRate()';

END order_pkg;
/
```