UNIVERSITÄT
BERN

Simple MLE Deployer

A simple (Web) Tool for exercising the Oracle Multilingual Engine

Bachelor Thesis

Julian Weyermann
from
Herrenschwanden BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

27. July 2020

Prof. Dr. Oscar Nierstrasz
Dr. Boris Spasojević

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Programming for relational database management systems with SQL and/or its various extensions and performing data analysis directly on the database is difficult. This comes from the fact that language support is limited. For example PL/SQL (in the case of Oracle) is not widely used and as a consequence there are not many libraries available for it.

Oracle aims to resolve this issue with the Multilingual Engine (MLE for short) which is an extension of their relational database management system Oracle Database. MLE enables developers to use JavaScript or Python along with the associated libraries.

However in its experimental state at version 0.3.0 of MLE the deployment of such JavaScript and Python code to the database is complicated and error-prone.

As the support for JavaScript is more mature than the support for Python, the goal of this project is to simplify the deployment process of JavaScript source code to MLE, enabling developers to try out MLE without the need to know a lot about its internals.

This is achieved by the creation of a small library which automates the deployment process almost completely. Alongside the library a website is available that interfaces with the library and further facilitates testing of MLE.

Acknowledgments

I would like to thank greatly Prof. Dr. Oscar Nierstrasz and Dr. Boris Spasojević of the Software Composition Group for making this bachelor project possible.

I would like to show my gratitude to Dr. Boris Spasojević in particular for his patience and precious hints. Without him I would not have been able to finish this project.

Last but not least, I would like to thank all the people around me who invested their time and effort to make this a better project.

Contents

1 Introduction

Directly analyzing or validating data stored in a relational database, i.e., without extracting it, is a difficult task. This comes from a lack of language support for programming languages that would support the study and analysis of data. Oracle aims to resolve that with the development of MLE as an extension to its relational database management system (RDBMS). MLE stands for "multilingual engine", which, as the name suggests, allows the execution of various programming languages. This is achieved by the use of an embedded virtual machine, GraalVM. This project aims to enable interested parties to try out MLE and use it more conveniently.

SQL stands for "structured query language" and is a query language, a type of domain-specific language, used to read, write and modify data primarily in the context of a relational database. It cannot be used to perform validation or complex analysis of data. To address that, Oracle implemented the extension language called PL/SQL to SQL, which enables the execution of imperative code in the RDBMS since SQL itself is purely declarative. At the same time, PL/SQL is a procedural extension that allows for a much more extensive range of functionalities to be programmed. Because of its niche, the community around PL/SQL is not large enough to supply frameworks and libraries which would allow data analysis or other tasks which require more complex code. Moreover, it is syntactically very different from most of today's widely used programming languages, which makes it inconvenient to learn and use.

This is where the Oracle Multilingual Engine (MLE) comes to play. It is an extension for the Oracle Database, which is at an early stage of development (as of time of writing) and is available as a preview in an experimental build of the Oracle RDBMS. In its state in version 0.3.0, it allows users to execute code written in JavaScript[1] directly on the database and apply it to the stored data without needing the data to leave the database. Running JavaScript gives developers access to a wide selection of publicly available libraries. Running libraries is one of the reasons why having the possibility to run commonly used programming languages is desirable in the first

[1] By JavaScript we refer to the standardized ECMA Script 2015 and later

place. The aim of MLE is that JavaScript can be used instead of PL/SQL, while PL/SQL acts as the connecting layer between MLE languages and SQL.

MLE is based on GraalVM[2], which is a polyglot virtual machine. GraalVM is embeddable into native applications like, in this case, the Oracle Database and allows executing a wide range of programming languages efficiently. As a result, there is the possibility of having multiple programming languages to choose from based on needs in MLE in the future. The latest release of MLE at the time of writing, version 0.3.0, emphasizes its support of JavaScript, while the deployment of Python code is possible too. However, for Python, fewer features are supported, and the documentation is rudimentary. This version was used throughout the work presented in this thesis.

Bridging the gap between the dynamic languages being executed on GraalVM and PL/SQL is a new and complicated process. It therefore requires many manual steps, such as loading the JavaScript code into a table inside the database, based on this making a so-called JavaScript source object inside the database and then finally making the JavaScript source available as PL/SQL function.

One particular challenge with JavaScript and PL/SQL interoperability is that JavaScript is dynamically typed (i.e., does not do any static or compile-time type checking), which conflicts with the statically-typed (PL/)SQL.

Because all those things have to be done manually and there is no tooling available, deploying a JS function to the database is time-consuming and error-prone.

In this thesis, we present a tool that automates the described process. It requires the developer to enter the code of the JavaScript function they wish to deploy as well as some metadata about that function. This includes the function name, type of arguments, and return value. Based on that, our tool, the Simple MLE Deployer, automatically deploys the code into the database and makes it available as a PL/SQL command.

To implement Simple MLE Deployer, we settled for an application written in JavaScript that runs in Node.js. We also provide a web application that provides a comfortable way to use the library which we developed over the course of this thesis. This web application is implemented using the npm package Express.js, which runs in Node.js too.

[2]https://www.graalvm.org/

2
Technical Background

In this chapter, we will present a deeper insight into the terms and technologies used in this project. In the section 2.1, we motivate our work by describing why developers might want to use MLE from a technical standpoint as well as one of the main issues with version 0.3.0 of MLE: the difficult and error-prone manual deployment process of JavaScript source code.

In section 2.2 it is explained what the Oracle Database is. In section 2.3 an overview of the procedural database language PL/SQL, which is used by Oracle Database, is given. section 2.4 gives an insight into what MLE is and how it integrates with the existing features of the Oracle Database and its imperative programming language PL/SQL.

In section 2.5, there is information on Docker, which is used to run the release 0.3.0 of the Oracle Database with MLE. Finally in section 2.6 some information is available on the software used to develop the Simple MLE Deployer.

2.1 Overview

The goal of MLE is to enable developers to use JavaScript in addition to PL/SQL based on their requirements. One of the benefits of this is that more developers are familiar with JavaScript than with PL/SQL. Also, there are many libraries and frameworks available for JavaScript, which enable users to reuse functionalities of all kinds. This also can be leveraged inside MLE. MLE is implemented by embedding GraalVM, a polyglot virtual machine, into the database. With GraalVM being polyglot, in theory, various programming languages could be used in MLE in the future.

Languages must be supported by MLE first. Among other things, an interface to (PL/)SQL has to be implemented. In version 0.3.0, this is mostly done for JavaScript and, to some extent, Python.

In the case of JavaScript, the biggest issue facing developers wishing to use MLE is that

(PL/)SQL is statically-typed while JavaScript is dynamically-typed. As a consequence, for one, there is no direct mapping between some of the data types of the languages (e.g., the `BOOLEAN` JavaScript datatype does not have an equivalent in the Oracle SQL standard). Secondly, for some data types, there are subtle differences, which may be an issue (e.g., the `NUMBER` datatype in SQL does not cover all the values of the JavaScript `NUMBER` datatype). This is why all JavaScript code must be wrapped inside a PL/SQL function to convert the types to and back from JavaScript to partially bridge that gap. Unfortunately, this makes deployment of code from dynamically typed languages to MLE somewhat inconvenient. The tool Simple MLE Deployer we developed in the process of this bachelor project makes this deployment process more manageable.

2.2 Oracle Database

The "Oracle Database" is a relational database management system that uses the declarative SQL as its query language. SQL can be seen as the industry standard for relational database systems. The details of SQL, however, vary among the various implementations of database systems. As an extension for tasks requiring procedural steps, there are the proprietary procedural extensions available on the Oracle Database known as PL/SQL.

The Oracle Database also supports a limited execution of Java as well as C code since version 8i. This was a first attempt at bringing other programming languages to the Oracle Database. It is, however, limited in terms of available APIs and is implemented entirely differently from MLE. This means that besides the goal of making the database compatible with more languages, the ability to run Java and C code on versions 8i to 19c has nothing to do with MLE and is thus outside the scope of this thesis.

The Oracle Database was first introduced as "Oracle V2" in 1979, and with it, the first predecessor of the seven years later released ANSI standard SQL, to which the Oracle Database mostly adheres today. The Oracle Database only got the word database in its name with the release "Oracle8 Database" in 1997. The latest version is 19c and was released in February 2019.

2.3 PL/SQL

PL/SQL is a proprietary programming language that can be used to implement more complicated behavior directly on the database than is possible with just SQL. SQL is purely declarative and used as the primary query language in Oracle Database. PL/SQL is statically typed, and its syntax is based on the Ada programming language [1]. The form of a PL/SQL function is either a so-called "stored procedure" (SP) or a "user-defined function" (UDF).

Stored procedures, as opposed to user-defined functions, never have a return value and thus usually are used to update the database in one way or another. A good example provided in the documentation of MLE (see section 2.4) is a stored procedure to implement a salary raise for employees whose data are stored inside the database. Stored Procedures are, in general, used for batch processing and updates of data.

On the other hand, user-defined functions always have a return value. They are typically used to make an SQL statement more precise. This means they can be used to make a select statement

more precise than was possible with just plain SQL. Developers can create custom functions additionally to functions that already exist in plain SQL like `AVG`, `MAX`, `MIN`, etc. An example could be to find all lines where an integer value lies in some interval. UDFs are commonly used in queries, the same way as built-in functions such as `AVG`, `SUM` or `COUNT` are. For a demonstration of the complexity of a PL/SQL user-defined function see Listing 1. This UDF can be called as it can be seen in Listing 2. An example for a stored procedure written in PL/SQL can be seen in Listing 3. This however has to be called differently, as seen in Listing 4. The last line before the `END;` prints the result to the internal cache of the database.

```
CREATE or REPLACE FUNCTION multudf(a IN number, b IN number)
RETURN number
IS
    result number;
BEGIN
    result := a * b;
    RETURN result;
END;
```

Listing 1: A simple UDF that calculates the product of two numbers

```
select multudf(4, 5) from dual;
```

Listing 2: The PL/SQL statement to call the function defined in Listing 1

```
CREATE or REPLACE PROCEDURE multsp(a IN number, b IN number,
result OUT number) IS
BEGIN
   result := a * b;
END;
```

Listing 3: A simple SP that calculates the product of two numbers

```
DECLARE
    c number;
    x number;
    y number;
BEGIN
    x := 4;
    y := 5;
    multsp(x, y, c);
    dbms_output.put_line(c);
END;
```

Listing 4: The PL/SQL statement to call the procedure defined in Listing 3

2.4 MLE

Oracle MLE stands for “Multilingual Engine”. It is an extension of the Oracle Database 12c. ML enables users to execute JavaScript and, to some extent, Python code. This is achieved by

the use of GraalVM, another project by Oracle. GraalVM is a polyglot and embeddable virtual machine that runs applications written in various languages such as JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Groovy, Kotlin, Clojure, and LLVM-based languages such as C and C++.

MLE aims to supplement PL/SQL with more modern and broadly-used languages. As there is an additional effort required to bridge the gap between the programming languages just mentioned and SQL, only JS and Python are supported as of right now. The support for JavaScript is much more mature compared to Python. The support for Python is highly experimental at this point and, therefore, outside of the scope of this thesis. For JavaScript currently (as of version 0.3.0 of MLE), there are only two direct type-mappings possible: The JavaScript `NUMBER` type is directly mapped to a `NUMBER` in (PL/)SQL. The JavaScript type `STRING` is mapped to the `VARCHAR2` type in (PL/)SQL. For other types, there are no direct conversions available. For example there is no equivalent in the Oracle (PL/)SQL standard for the the JavaScript type `BOOLEAN`, the reason being a design decision at an early stage of the Oracle RDBMS by the teams at Oracle. A JavaScript `BOOLEAN` would, therefore, for example, have to be stored as `VARCHAR2` in the database and be parsed when reading it back into JavaScript.

While MLE runs JavaScript program code, it is worth noting that it is not a Node.js environment. Access to the local file system is one of those things, which is not possible in MLE (as there is no file system inside a Database) as well as network access, which is not possible either with version 0.3.0 of MLE.

The version 0.3.0 of MLE is based on Oracle Database 12c and available for download in the form of a docker image on the Oracle Technology Network[1].

2.5 Docker

Docker is a virtualization environment that enables users to run a specially prepared container independently of its underlying environment. Containers allow developers to isolate their apps from the environment, as a container is a standardized software unit that always provides the same environment no matter where it is executed. The fact that the Oracle Database with the MLE extension is shipped as such a container makes distributing, running, and working with Oracle MLE convenient as there is not much work needed to set it up. However, in our experience, the docker image was sensitive about the environment it was running in despite the claims of Docker about separating the image environment from the host.

We can confirm that the Docker image of MLE works reliably on Oracle Linux 7.7 while not working reliably on Linux Mint 19.2 and 19.3 and Ubuntu 18.04. We assume that this is because Docker shares the kernel of the host to some extent with its containers, resulting in different behavior on different kernel versions. However, the inner workings of Docker are not part of the topic of this thesis.

[1] http://www.oracle.com/technetwork/database/multilingual-engine/overview/index.html

2.6 Node.js and Node Package Manager (npm)

Node.js is a JavaScript runtime. It enables the execution of JavaScript code outside of a browser, where JavaScript is typically executed and was originally designed for. We use the possibilities of Node.js and npm in Simple MLE Deployer to run the web server and deliver the website. The library for the deployment of JavaScript code is written in JavaScript too. This means it runs in Node.js, away from the context of a web browser, too.

By default, Node.js uses the V8 engine, which was initially implemented for the Chrome browser. Alongside Node.js, the node package manager, abbreviated to npm, is available, which provides an easy way of installing libraries and frameworks written for Node.js. As a consequence, Node.js is a convenient, quick, and efficient environment for developers to develop web and other applications that are not designed to be run in a browser environment.

Node.js and npm can be installed using the Node Version Manager (nvm). It is a convenient tool for handling different versions of Node.js and npm for different projects and resolves permission issues that might arise when installing packages with npm.

3 Simple MLE Deployer

This chapter presents our solution to the issue that the deployment process for JavaScript code is error-prone because it involves too many manual steps that have to be taken. The procedure for manually deploying JavaScript source code is described in section 3.1.

The Simple MLE Deployer is implemented as a library written in JavaScript, which deploys a given JavaScript function to MLE, accompanied by a simple web application that allows for a convenient way to use the library. A graphical representation of the architecture of Simple MLE Deployer can be seen in Figure 3.1.

The base of the library is a JavaScript function which requires JavaScript code alongside some metadata as arguments. It generates and executes all the PL/SQL commands required to deploy the JavaScript code to the database. This automates the deployment resulting in a function callable directly from SQL. Based on this library, we further created a web application. It relieves the user from reading up on the internals of MLE and on the deployment process of JavaScript to the database and offers a first experience with MLE instead.

Through the use of the Web application, users can get an insight into what can be done with Oracle MLE by being able to write and deploy JavaScript. Also, the user can access and execute it through the provided option of running SQL queries on a database with MLE support.

3.1 Manual Deployment

This section describes how the manual deployment of JavaScript Code as a UDF is done. There are various ways to deploy source code to MLE. They are all the same in terms of performed steps but differ in some details, mainly in terms of how the JavaScript code is loaded into the database. In this section, we present how a user typically would deploy a JavaScript source by hand. It is essentially the manual process that the Simple MLE Deployer automates. The library

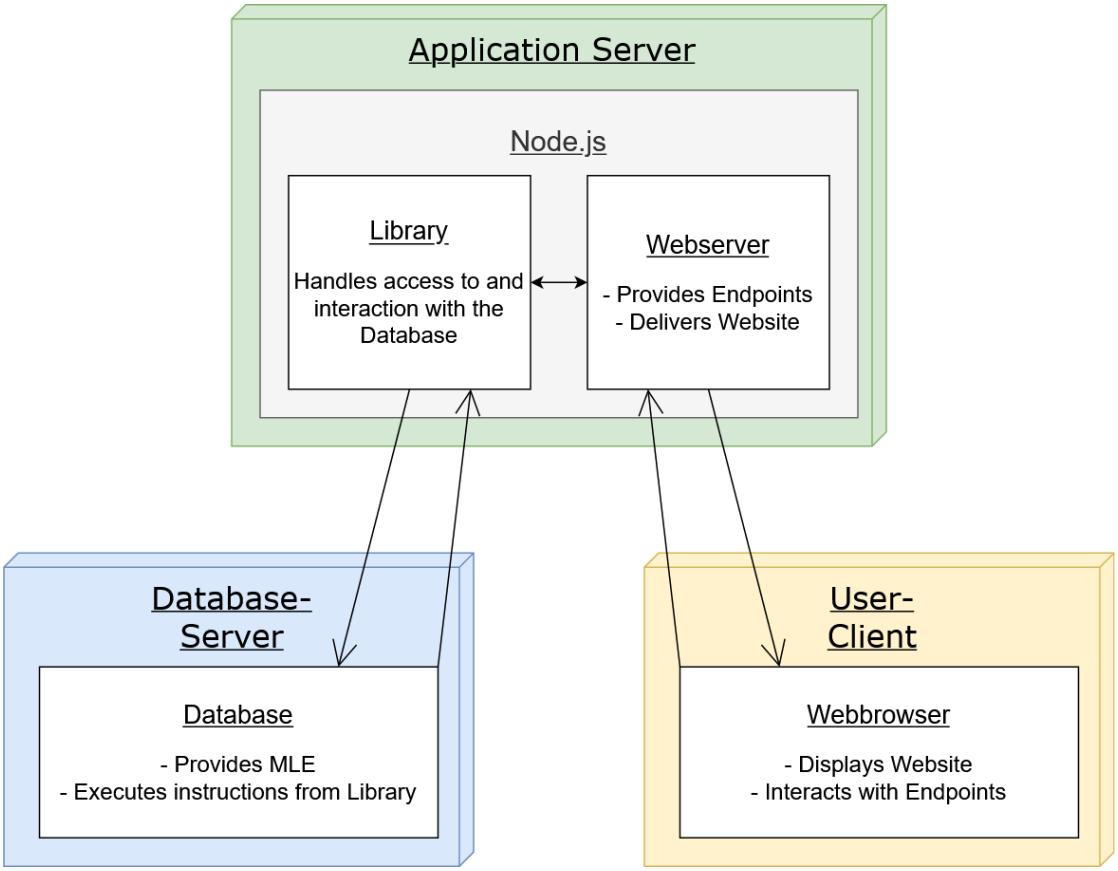

Figure 3.1: Structure of Simple MLE Deployer

we developed takes a script provided in the Documentation[1] of MLE as starting point.

We use a simple greeting function as our running example for demonstrating the manual deployment process. The function takes a `STRING` "name" as argument and returns a `STRING`, that greets the name given as input argument. The function is shown in Listing 5.

```
module.exports.helloWorld = function (name) {
  return "Hello " + name;
}
```

Listing 5: The JavaScript source code that should be deployed to MLE

It is possible to deploy multiple JavaScript functions that depend on each other, but for simplicity, we will focus on one for our running example. If there are multiple dependent JavaScript functions, it however is not possible to make multiple of them available as callable PL/SQL function at once.

The JavaScript source code from Listing 5 needs to be loaded into a CLOB field in the database. CLOB stands for "character large object" and stores Unicode encoded strings of up

[1] https://oracle.github.io/oracle-db-mle/releases/0.3.0/js/udf/

to 2,147,483,647 characters length. The loading of the source into the database can be done in various ways. One way is to copy the JavaScript code and store it to a table as is done in Listing 6. The same command can be used from a JavaScript Script, as seen in Listing 7. In this case, the values that are inserted using the SQL command, can be replaced with JavaScript variables as it is done in Listing 7.

```
INSERT INTO mylobs
VALUES (1, 'module.exports.helloWorld = function (name) {
  return "Hello " + name;
}')
```

Listing 6: SQL code to load the JavaScript code into the database manually

```
INSERT INTO mylobs
VALUES (:id, :data), {id: 1, data: data}
```

Listing 7: SQL code to load the JavaScript code into the database from JavaScript. The code is stored in the "data" variable.

The `mylobs` table has two columns: The id, which is a unique number, as well as a CLOB column for the JavaScript source as a UTF-8 encoded string. This table is temporary and must be created manually before the deployment procedure and may be dropped afterwards. It is also possible to omit dropping it and instead increase the `id` value in Listing 6 and Listing 8 every time new JavaSource code is deployed.

Next, from the JavaScript stored in the CLOB field, as seen in Listing 8, a so-called JavaScript source must be generated. A JavaScript source is the internal representation of the JavaScript source code in the database from which finally the PL/SQL function can be created.

```
CREATE OR REPLACE JAVASCRIPT SOURCE NAMED "mysource" USING CLOB
SELECT data
FROM mylobs
WHERE id = 1
```

Listing 8: SQL code to create a database internal representation of the JavaScript Code

Finally, based on the source object, the JavaScript function can be made available as a UDF (or SP, if there is no return value). The function from our running example must be a UDF because, as explained in section 2.4, it returns a value. At this point, PL/SQL is used to wrap the JavaScript function in a PL/SQL function for bridging the gap between the statically-typed SQL and dynamically-typed JavaScript. This is done with the statements as seen in Listing 9.

```
CREATE OR REPLACE FUNCTION helloWorld(name IN VARCHAR2)
RETURN STRING AS
LANGUAGE JAVASCRIPT NAME 'mysource\.js.helloWorld(name string) return string';
```

Listing 9: SQL code to create the callable PL/SQL function from the internal representation of the JavaScript source code

The UDF then can be called with the code shown in Listing 10. The result of the query of Listing 10 is shown in Listing 11:

```
SELECT helloWorld('Rainer') from dual
```

Listing 10: SQL query which calls the newly defined PL/SQL function that is based on the JavaScript code

```
"Hello Rainer"
```

Listing 11: result of the SQL query in Listing 10

3.2 Library

The core of Simple MLE Deployer is the library written in JavaScript. It generates the PL/SQL code that is required to make a JavaScript function available as a PL/SQL User-Defined-Function or Stored-Procedure as described in section 3.1.

The library generates the PL/SQL code based on the metadata of the JavaScript function, the JavaScript code itself and data required for the database. As an example for the code and the metadata, we use Listing 5, where the relevant parts are color-coded along with the corresponding description below.

```
module.exports.helloWorld = function (name) {
  return "Hello " + name; }
```

- functionName: The name of the JavaScript function which should be exposed and callable as PL/SQL function. In our running example in section 3.1, this would be `helloWorld`, used in Listing 9.
- sourceCode: The JavaScript code, written by the developer, as in Listing 5.
- argArray: An array of Objects with the name and JavaScript variable-type of all arguments the function takes. The type may be either `STRING` or `NUMBER`. An example of such an array can be seen in Listing 12
- returnType: The type of the value returned by the JavaScript function. May be `STRING` or `NUMBER` as for the elements in the argArray.

The data that is used by the library but is not JavaScript code related is the following:

- dbInstance: This Database Connection String defines into which database instance the code is deployed. This of course needs to be an instance with MLE. An example could be `140.238.173.160/ORCLPDB1`.
- dbUser: The username which should be used to login to the database specified in the dbInstance string. The default set by Oracle for demonstration and exploration purposes is "scott".

```
{
  "argArray": {
    "1": {
      "argName": "username",
      "argType": "STRING"
    },
    "2": {
      "argName": "password",
      "argType": "STRING"
    },
    "3": {
      "argName": "age",
      "argType": "NUMBER"
    }
  }
}
```

Listing 12: An example for an Argument Array taken by the library

- dbPwd: The password to authenticate the user specified in the username string. The default set by Oracle for the user "scott" is "tiger".
- moduleName: The name of the module as which the code should be represented. This from our experience can be quite arbitrary and is irrelevant from a users perspective. To the database the moduleName matters as it defines under what name the JavaScript source will be stored. It is not possible to have two modules with the same name. An example would be `mysource` used in Listing 8 and Listing 9.

The module-, function- and argumentName may be anything which does not violate (PL/)SQL or JavaScript naming conventions (e.g. function names like `ADD` or `function` are not allowed). The library does not check that, however the database will check at execution time and throw an error when the user tries to call this function.

Regarding the deployment of multiple (dependent) functions at once, this is possible, however, only one at a time can be deployed as a PL/SQL function.

In case there are multiple functions that need to be deployed, which are potentially split up across multiple files, the source code maybe packed with "WebPack". WebPack is an npm package that resolves dependencies of Node.js code and compiles the source code and its dependencies into one big JavaScript file. Having all dependent code in one source file is a requirement for the deployment to MLE.

While the other steps like loading the code into the database and creating a database-internal representation from it were quite trivial to automate, the automatic creation of the PL/SQL function as in Listing 9 was not easy to achieve.

3.2.1 Generating the required PL/SQL code to deploy the JavaScript code

In general the deployment process performed by the script is the same as presented in section 3.1. However we would like to highlight some particularities in this section.

One thing that differs quite a lot from the manual procedure, is that the library uses a table called `js_sources` to store the source code (as it is done in Listing 6) anytime something is deployed. It assumes that table already exists upon deployment. This table is so that it automatically increases the id of the source code. This id value is stored in a variable so that the internal JavaScript source can be created based on that id, as done in the manual steps in Listing 8.

Another thing worth mentioning is that the library uses two different techniques to generate the SQL commands. Whenever possible it uses a binding between the JavaScript variables which should be placed in the PL/SQL command. A binding maps a JavaScript variable to a (PL/)SQL variable or vice versa. This is however only possible when inserting values like the JavaScript code into the sources-table. Creating PL/SQL code dynamically with bindings however is not possible. As a secondary technique to generate SQL commands, the library therefore uses JavaScript string concatenation to generate the necessary PL/SQL, such as the commands in Listing 8 and Listing 9.

3.3 Web Application

The web application wraps the library of Simple MLE Deployer and provides a web server and a website for a smooth user interaction with the library and therefore MLE.

3.3.1 Web Application Endpoints

The Simple MLE Deployer web server provides two endpoints. One for deploying JavaScript code to the Database with MLE (submitJs) and one for running SQL (submitSql).

3.3.1.1 submitJs

Simple MLE Deployer defines the `/submitJS` endpoint which is located at the root of the website and accepts `POST` requests. The payload of the request must be delivered in JSON format and must include all data required by the library of Simple MLE Deployer. This data is explained in section 3.2. The endpoint extracts the data from the JSON and invokes the library described in section 3.2 with the given data, which then deploys the function based on the input parameters. The endpoint itself does not provide any form validation, as the website will validate the inputs, and the goal of the Simple MLE Deployer is that users can try out MLE with full access to the database in any case.

The endpoint sends the response in the JSON format. In case of success, it sends `statusCode: 1` along with the success message “Code deployed successfully”. If an error occurs during deployment, `statusCode: 0`, along with the error message is sent as response.

3.3.1.2 submitSql

The other endpoint that Simple MLE Deployer defines is also located at the website root and can be called at `/submitSql`. Just like the `/submitJs` endpoint it accepts `POST` requests without restriction and requires payload in the JSON format. The JSON data must include four parameters:

- sqlCode: The SQL query that should be executed on the database
- dbInstance: The Database on which the SQL query above should be executed on
- dbUser: The username for which the SQL query should be executed.
- dbPwd: The password for the dbUser

The endpoint does not validate the input data, as the website does the validation of the inputs, and the users of the website need full access to the database. Also, the security of the application is outside the scope of the thesis. If the SQL code is flawed or the database parameters are wrong, the function or the database is going to return an error, which makes server-side validation obsolete for our use case. The endpoint extracts those values from the JSON data and

1. Establishes a connection to the server with the given database string and the credentials,
2. executes the provided SQL query,
3. stores the result, that the database returns on the SQL query,
4. closes the connection to the database
5. and returns the stored result of the SQL query.

If the operation is successful, the endpoint responds with a JSON object containing the database result. If an error occurs, the error message is sent as the response in the JSON format.

3.3.2 Web Application User Interface

Figure 3.2 shows a rendering of the user interface of the Simple MLE Deployer. It's a picture of the user interface with four colored rectangles added to differentiate between the areas. In the red area, described in subsubsection 3.3.2.1, the users can enter the JavaScript code, which they want to deploy. In the green area, described in subsubsection 3.3.2.2, SQL queries can be written, for example, to test a JavaScript function, that was deployed as PL/SQL function. The yellow area is where the results of the SQL query are displayed. More on that can be found in subsubsection 3.3.2.3. In subsubsection 3.3.2.4, the settings and configuration area, marked in blue is described in more detail.

We built the interface using the lightweight CSS framework "Bulma"[2], which provides so-called tiles among many other things, that create spaces for the various colored areas described in this section. The user interface is delivered through a web application, that is based on the Node.js framework Exress.js.

[2]https://bulma.io/

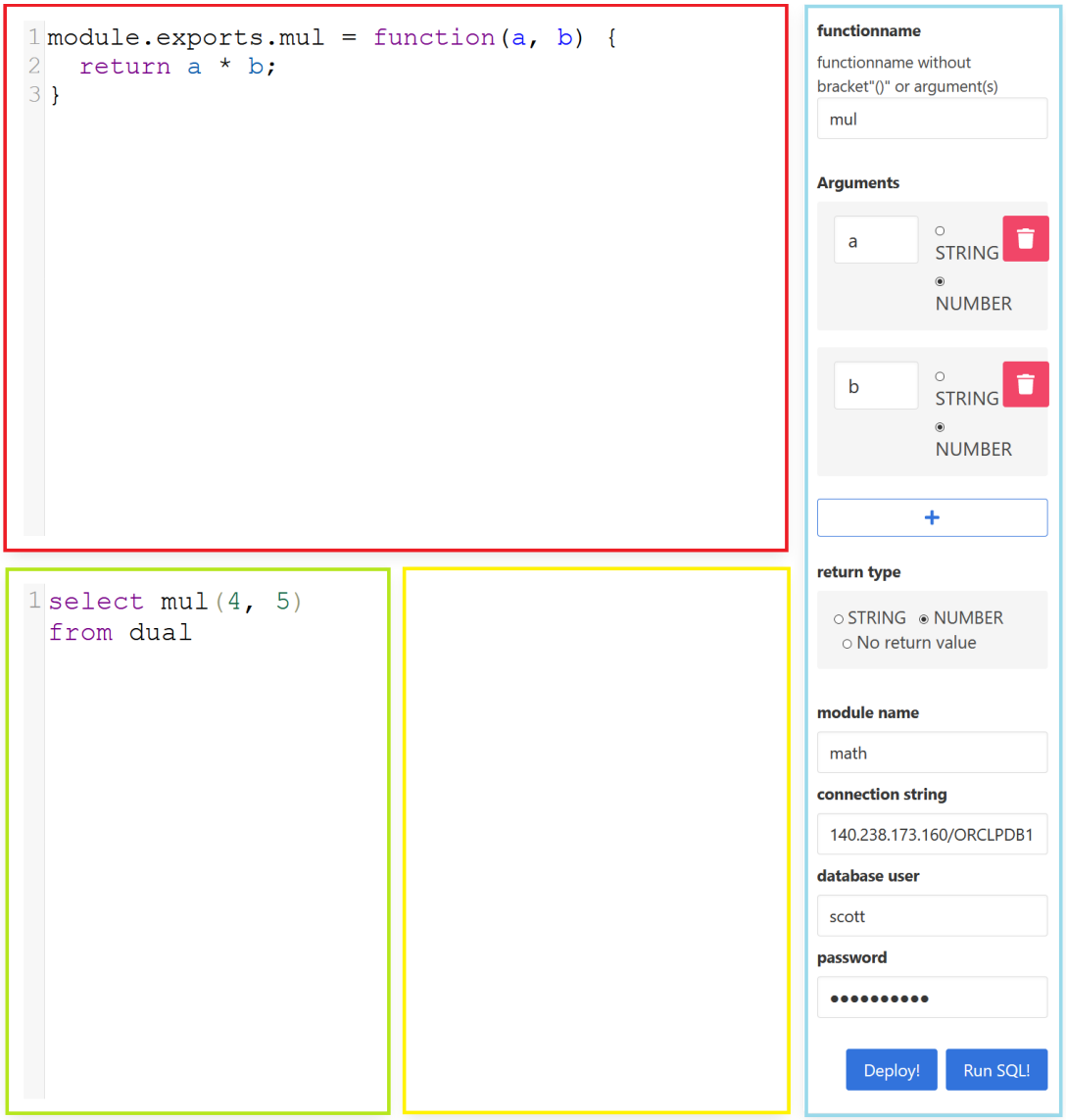

Figure 3.2: The web application for writing and deploying code to MLE

3.3.2.1 JavaScript Development Area (Red)

The red area in Figure 3.2 enables the users to write basic JavaScript code, that is to be deployed to the database. The text editor area is implemented using the library CodeMirror[3], which is a versatile text editor written in JavaScript. It supports syntax highlighting, auto-completion, and many other features that are present in common modern code editors. Those qualities make writing code in a browser window more inline with developer expectations. For the JavaScript

[3]https://github.com/codemirror/CodeMirror/

area naturally JavaScript syntax highlighting is enabled.

3.3.2.2 SQL Query Area (Green)

The green area shown in Figure 3.2 enables the user to write SQL queries according to the standards of the Oracle Database directly from the browser. This area intends that users can call the PL/SQL function that they wrote in the JavaScript Area above and have deployed. This part of the website again uses a CodeMirror instance, this time with syntax highlighting enabled for SQL.

3.3.2.3 SQL Result Area (Yellow)

The yellow frame in Figure 3.2 shows the area that is reserved space for displaying the result of the SQL query. The result is displayed in the raw JSON format like the endpoint described in subsubsection 3.3.1.2 provides it.

3.3.2.4 Settings and Configuration Area (Blue)

In the area on the right side of the screen, we can identify three distinct sections. In the first section of Figure 3.3, the fields for the required metadata for deploying the JavaScript function are present.

In the second section of Figure 3.3 the fields for entering the database parameters are present. This information is necessary to connect to a database and is used by the two buttons below (third section of Figure 3.3). The first one, labeled "Deploy!" uses the information from the JavaScript metadata fields and the connection details of the database to deploy the code written in the JavaScript development area to MLE. It uses the `submitJs` endpoint which is provided by the web server (described in subsubsection 3.3.1.1). The second one, labeled "Run SQL!", only requires the database details to run the code written in the SQL development area on the given server. To do this, the `submitSql` endpoint provided by the web server is used. More details about this endpoint can be found in subsubsection 3.3.1.2.

When clicked, both buttons give immediate feedback by displaying a turning circle. Depending on if the request to the endpoint was successful, the buttons either turn green for a fraction of a second or red until the alert is confirmed. If the request was unsuccessful, the error message from the library is displayed as a JavaScript alert on the website.

In case of success after the "Run SQL!" button was clicked, the result of the SQL query is displayed in the reserved area described in subsubsection 3.3.2.2.

3.3.3 Limitations

It is clear that the website is not suitable for big projects. It is however, not intended to use for large projects but for trying out MLE and trying out throw-away code. Surely an extension of the features would be possible. We do not think this would be sensible, though, and an integration with an established IDE for JavaScript would be much more beneficial for more significant projects.

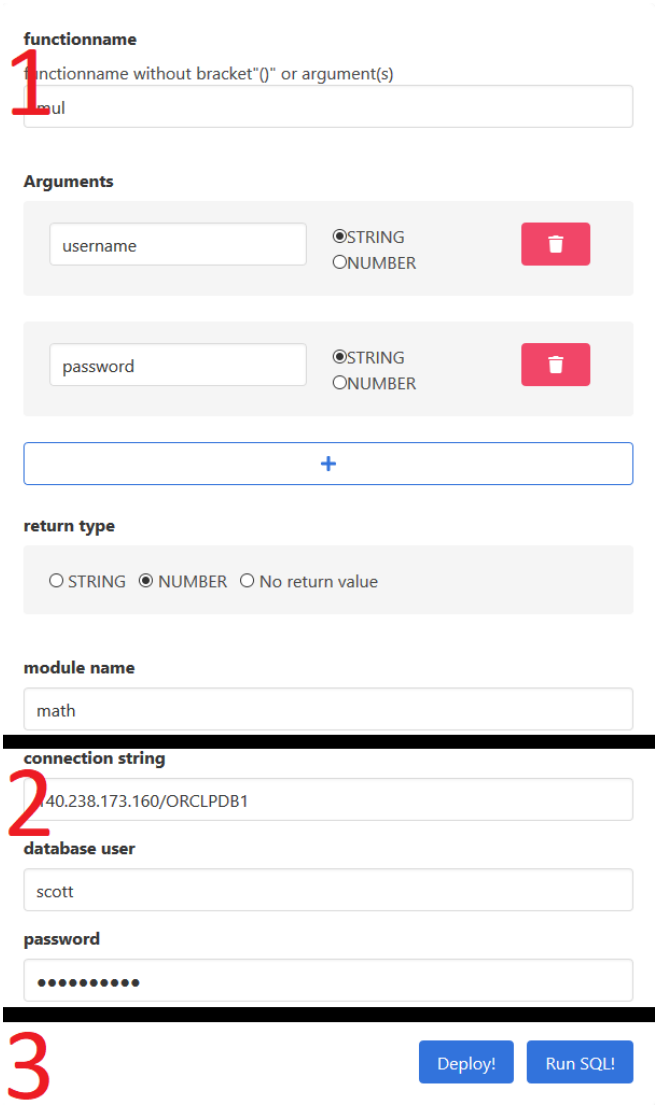

Figure 3.3: Metadata of the JavaScript Function, Connection Details and execution Buttons

4 Challenges

The original objective of this project was to generate a report PDF directly inside the database using data stored in the database. We did not succeed at that and had to rescope and developed Simple MLE Deployer instead, for the reasons described in this chapter.

4.1 New technologies

New technologies often have the same issues. The documentation is not mature, there is a lack of tooling, and comunity posts online are not much of a help when one runs into issues or gets stuck. MLE is no different.

The documentation at the time of writing often uses examples to show functionalities, and when trying them, they generally work. Unfortunately, the reader does not learn why or how they work. Moreover, even more importantly based on this documentation, it often is impossible to find out why some error occurs, where its origin lies and what it means. The biggest reason for this is that as described in chapter 2, the JavaScript code is wrapped inside either a PL/SQL stored procedure or user-defined function. This often results in errors with a PL/SQL "ORA" error code and a short cryptic error message, which most of the time does not have to do anything with the actual problem.

This leads to the problem of looking for solutions using a web search engine. With Google, we were able to find most of those error codes. However, the provided solutions only address issues with PL/SQL code. This, in our case, rarely was helpful as there is minimal PL/SQL code present, which just wraps around the JavaScript code. The reasons behind the minimal help of Google are obvious: Almost no one uses MLE so far, and if someone does, it is only for experimental purposes as MLE is not yet available for production environments because of its early development state. All of this leads to frustrating debugging sessions with much trial-and-error.

4.2 MLE is not Node.js

The documentation seems to suggest that Node.js libraries can be used in MLE, just like one does when working with Node.js with just one additional step: Webpack. This means libraries first can be installed using the Node Package Manager (npm) with the following command in bash:

```
npm install trigonometry-calculator
```

The package trigonometry-calculator then can be used conveniently in any JavaScript source-file by just inserting this at the top of the file:

```
var calculator = require('trigonometry-calculator');
```

The functions of the library then can be called with:

```
calculator.function();
```

With the JavaScript file finished and saved, developers then have to use WebPack to resolve dependencies in the code and pack everything into one big JavaScript file. WebPack is an npm library too, which means it can be installed the same way as in the example before. This command installs the webpack-command-line-interface along with the main package and makes it available to all Node.js projects on the local computer with the option `-g`.

```
npm install webpack webpack-cli -g
```

WebPack then can be used to pack the code like so:

```
npx webpack-cli --entry=./calculator.js --output=./calculator_packed.js
    --output-library-target=commonjs
```

The npx command enables users to execute npm packages directly from the command line if they can be used this way. The `--entry` option points to the file, which includes pointers to the node dependencies, and which was created by the developer. The `--output` option designates the file path where the packed file (i.e., the result of the webpacking process) should be stored. The `--output-library-target` option defines for which of the supported languages (by webpack) the output file should be suitable.

However, when we tried to deploy the file that resulted from the WebPacking-process, we were not able to use functions from the external libraries defined in the original JavaScript file sometimes. For instance, the example in this section works, while on the other hand we always had issues with libraries that would generate PDF files. We were not able to evaluate why this was the case. Nevertheless, there are quite a few differences between running JavaScript in a browser and Node.js. The most significant difference may be that there is no window object, which many npm packages rely on heavily. This also applies to jsPdf. There are possibilities to mock that object[1], but some elements still seem to be lacking, which are needed by the module to run.

[1] https://stackoverflow.com/questions/47791928/jspdf-referenceerror-window-is-not-defined

4.3 Caching Issues

During development and testing, it happened quite frequently that when attempting to change and redeploying an existing function, the changes did not take effect, even though no errors occurred. This behavior was underlined by much shorter deploying times than when the changes took effect. This seemed like a difficult to resolve caching issue. As we worked with UDFs solely for debugging purposes, we never came across the following quote in the documentation for the SPs:

> *Right now, MLE caches the source code internally so that the old code of simplesp.js will be executed when the procedure sayHello is called. For this reason, we need to flush the internal code cache before calling the procedure.*[2]

We only came across this text by chance, much later, after changing the scope of the project, when it did not matter anymore. However, this gave us the chance to implement the solution to the issue into the library. It executes `call dbms_session.reset_package();` to flush the cache after a function has been (re-)deployed.

4.4 Documentation

The documentation is not extensive enough, and the examples given are often not explained in-depth enough or not chosen wisely. For example, during the creation of the library, it was very unclear whether the marked red type in Listing 13 was a JavaScript or a SQL type because in all but one examples the type `NUMBER` was used which could be both, JavaScript and SQL. From that `STRING`, it became clear that it should be JavaScript. However, when we tried substituting this `STRING` with `VARCHAR2`, the function was still callable and worked as expected.

```
CREATE OR REPLACE FUNCTION helloWorld(who IN VARCHAR2)
    RETURN STRING AS LANGUAGE JAVASCRIPT NAME
    \'mysource\.js.hello(who string) return string\';
```

Listing 13: SQL code to create the callable PL/SQL function from the internal representation of the JavaScript source code

[2] Source: https://oracle.github.io/oracle-db-mle/releases/0.3.0/js/storedprocedures/

5 Future Work

In this chapter, we summarize what people looking to create future extensions and developments around Oracle MLE should include and consider doing. Some of our proposed future work requires changes to the MLE extension too, which has to be implemented by the teams at Oracle.

5.1 Tooling

As discussed previously (i.e., in chapter 4), there is no tooling available for interaction with MLE except for this project. As a result, the field for future work is wide open.

As MLE in version 0.3.0 requires all dependent code to be in one file, the use of WebPack or a similar tool, that packs the source code and its dependencies into one file is often required. Because the web interface is not suitable nor intended for bigger projects, the development of a plugin for an advanced text editor (e.g., Atom, vim, etc.) or even a feature-rich IDE like Webstorm or Eclipse would be beneficial. To our eyes, using the library from the command line alongside with WebPack still involves too much manual work, which could be automated.

With a simple wrapper that reads a JavaScript source code file into a String, however, the library of Simple MLE Deployer can be used as a command-line tool for deploying to MLE, enabling developers to use their preferred way of developing JavaScript code. An alternative would be to just change the following line in the code of the library:

```
// read JavaScript sourcefile. Kept the line for being able to
// switch between the sourceCode and filename with source code easily:
// If the name of file should be used, read code from it with
// "const jsSource = fs.readFileSync(sourceFile, 'utf-8');"
const jsSource = sourceCode;
```

Future work might also include the automatic recognition of some of the parameters required by the library. In this case the developer could select a JavaScript function, which should be

available in PL/SQL, the library then will evaluate the input arguments, and their type and the same goes for the return value. However for dynamically typed languages, this is impossible to completely and reliably achieve with static analysis, which means in case of JavaScript and Python an execution of the code previous to the deployment would be mandatory to find the needed data. Also, the module name could be based on the filename or some similar solution.

Besides that, the library may be extended in a way that it allows for making multiple JavaScript functions available as PL/SQL functions at once.

5.2 (Node) Library Deployment

In order for MLE to realize all its advantages over classic PL/SQL while using it, it is necessary that developers can use libraries in a comfortable and user-friendly way. Therefore the library handling in the development process with MLE should be extended so that the work with libraries is just as seamless as with Node.js. Preferably, the handling of libraries would be the same as with Node.js, as that is what most developers are used to nowadays.

The web application does not support this. However, the library can be used to deploy any JavaScript code. As it is required to have all connected JavaScript code in one file by MLE, this should not be difficult to achieve with the use of WebPack. WebPack is a JavaScript library that resolves and repacks Node.js dependencies into one JavaScript file. MLE requires this because it does not support dependent JavaScript source code files as of now.

However, the use of npm libraries also has to be supported reliably by MLE itself, which only can be achieved by the developers of MLE at Oracle and is not the case for the version used for our project. Compressing code from some libraries together with self-written code, deploying the compressed code to MLE, and using it there sometimes works flawlessly, but unfortunately, with some npm libraries executing the deployed code fails for various reasons.

We suspect it might have to do with differences between the environment. Some libraries are designed for use in the browser, and some for execution in Node.js. The circumstances differ between the environments: While there is a window object available in the browser, there is no access to the file system. In the Node.js environment, it is precisely the opposite way around. In MLE, neither a window object nor access to a file system is possible.

5.3 Type Conversions

In the future and for extended use, it will be beneficial if the MLE and the library grants its users more flexibility with type conversions. As it can be seen in Table 5.1[1] only three direct type conversions between JavaScript and (PL/)SQL are possible with the 0.3.0 release of MLE. The library uses two of them without allowing the user to change them, namely `NUMBER` to `NUMBER` and `STRING` to `VARCHAR2`. For convenience reasons and sufficient properties of we used `NUMBER` for Simple MLE Deployer. This can be easily switched over to `BINARY_DOUBLE`, if needed though.

To make the library more versatile, e.g., options for storing JavaScript `BOOLEAN` variables as (PL/)SQL `VARCHAR2` will be necessary.

[1] https://oracle.github.io/oracle-db-mle/releases/0.3.0/js/conversions/

This, however, introduces new issues. If, for example, a `BOOLEAN` is stored as `VARCHAR2` in the database (as there is no boolean type for the database), when reading it back from the database, the value is of type `STRING`. It then must be converted to a `BOOLEAN` in JavaScript again. So the conversions apart from the ones done by the library are not "symmetrical" and thus require additional code to behave as expected. Additionally, a `BOOLEAN` could also be stored as `NUMBER` in the database. Handling of those issues would have to be done by the developer.

It is unlikely that more direct conversions will be made possible by MLE in the future, as the SQL types are defined by the RDBMS and MLE has no influence on that. However it could be possible, that MLE stores for example a `BOOLEAN` as `NUMBER` in the future and automatically converts it back to the JavaScript `BOOLEAN` type in the future. This is speculation though.

	NUMBER	BINARY_DOUBLE	VARCHAR2
Number	direct, may overflow	direct	toFixed()
Boolean	0 or 1	0 or 1	TRUE or FALSE
String	error	error	direct
null	NULL	NULL	NULL
undefined	error	error	error
Object	error	error	error
Symbol	error	error	error

Table 5.1: Possible type conversions from the JavaScript to Database types

5.4 Automatic Recognition of metadata

In the future, another feature that might become handy is if the library could determine the metadata of the given JavaScript function by itself, so that the user just has to hand over the function name of the JavaScript function which he wants to be deployed, and the library determines all other parameters by itself. This would further reduce the possibility of user error.

The library determining the parameters on its own, however, would introduce new issues. Determining the appropriate variable type and conversion is not possible without parsing and running the JavaScript code. Even if the conversions could be determined automatically, there are still various data types from both languages which cannot be converted to the other language.

5.5 Deployment of multiple functions to MLE

A flaw of Simple MLE Deployer lies in the fact that only one JavaScript function can be made available as (PL/)SQL function at once. It is possible to have multiple functions which are depending on each other in the JavaScript code. However, only one of them will be callable from SQL queries. In most cases, this may not be an issue, as structuring the JavaScript code is possible (at least as long as it is in one file). However, for example, if all functions from the Node.js validator-module, which is used in the documentation by Oracle as an example, the user

would have to manually make the functions by creating PL/SQL calls like the following one for each function, which again is time-consuming and error-prone.

```
CREATE OR REPLACE FUNCTION isEmail (address VARCHAR2)
    RETURN NUMBER AS LANGUAGE JAVASCRIPT NAME
    'validator.js.isEmail(a string) return number\';
```

This issue could be solved by adding a loop to the library and changing the data structures in the future. Due to time constraints, this, unfortunately, is outside of the scope of this thesis.

5.6 Documentation

The documentation for MLE, at the time of writing, mainly consists of examples. On the one hand, this helps with getting started and finding the way around in MLE. On the other hand, those examples did not always work as expected. Furthermore, there is too little additional information given on what the examples exactly do or how they work precisely. So it is up to the user to guess and try out until he has a working example. Then there is still the uncertainty why it works, as there are so many variables which influence the result. This uncertainty is also due to the cryptic error messages the users get shown when trying to deploy or run code. During the manual deployment, errors often looked like this:

```
ORA-06575: Package or function PDFOUT is in an invalid state
06575. 00000 -  "Package or function %s is in an invalid state"
*Cause:    A SQL statement references a PL/SQL function that is in an
           invalid state. Oracle attempted to compile the function, but
           detected errors.
*Action:   Check the SQL statement and the PL/SQL function for syntax
           errors or incorrectly assigned, or missing, privileges for a
           referenced object.
```

This message is not helpful at all. There is no list of possible error messages present in the documentation, which makes finding the problem with error messages that do not tell the developer anything about the issue virtually impossible. Only by chance we figured out that JavaScript errors are denoted as such, as you can see in the following example:

```
MLE: SyntaxError: SCOTT.\"math.js\":1:120 Expected ; but found
b\n(function () { var exports = {}; var module = {exports: exports};
module.exports.kurz = function(a, b) {return a b;};
```

Also naming mismatches between MLE and JavaScript are denoted as ReferenceError:

```
Failed to run SQL Code:
failed to execute code: ORA-06550: line 1, column 117:
\nSCOTT.\"math.js\": ReferenceError: c is not defined
```

This leads to the assumption that the first error message presented here has something to do with the deployment process.

In general, the documentation is great to get started, but not usable if a developer wants to work with MLE in a productive environment. The reason for this is that the necessary information can only be found by reading the complete documentation because the necessary information could be anywhere in the documentation. Moreover, the documentation also is not searchable.

5.7 Support for more Languages

As the deployment of JavaScript and Python code is very similar, the deployment process likely will be similar for other supported languages in the future. As there is only Python available as another language for MLE version 0.3.0, the library could be extended to enable the deployment of Python source code.

6 Conclusion

After exploring the possibilities of Oracle MLE by attempting to generate PDF files directly on the database, we had to rescope the topic of this project to developing tooling. The lack of tooling made this decision easy and allowed us to discuss the main shortcomings of the 0.3.0 release of Oracles Multilingual Engine. It aims to bring more and broader used programming languages to Oracle relational database management system “Oracle Database”.

Based on these findings, we were able to design and develop a library that simplifies the process of deploying JavaScript code to MLE without prior knowledge of either MLE or its library. We created a website that facilitates the usage of the library, which in turn simplifies the interaction with MLE.

The Simple MLE Deployer uses a modular design, where the web application, the library for deployment and the code for running SQL on remote databases are independent of each other and it provides interfaces for other applications too. Therefore it is easy to extend and modify parts of the project in order to make it suitable for various use cases, including for example the usage of just the library from the command line.

On our way to a working piece of software, we learned that a extensive and precise documentation along with examples is invaluable and crucial for the correct usage of programming interfaces. The development of this tool also showed that PL/SQL does not have much in common with modern programming languages and that for this reason MLE can be highly beneficial to developers who are not familiar with PL/SQL.

A

Anleitung zum wissenschaftlichen Arbeiten

In this document, we provide all the required information on how to get Simple MLE Deployer running. In section A.1, all information required to run the project on a local machine is available. It is segmented into three parts: in subsection A.1.1, there are instructions on how to obtain the source code of Simple MLE Deployer. In subsection A.1.2, you can find the necessary instructions to get the docker image of the Oracle Database with MLE running on a local machine with Oracle Linux version 7.7. This Oracle SQL Database instance with MLE is not part of the thesis; however, it is crucial for demonstrating our work. Furthermore, in subsection A.1.3, the steps to deploying the Simple MLE Deployer application are listed. The instructions for running this Node.js application are tested for Linux Mint 19.2 and Ubuntu 18.04 machines. The application provides the website for deploying to the MLE Database, which enables users to try out MLE in a straightforward manner. As a result, people with no experience with setting up Node.js and Docker and all other tools required can still try out MLE. Also, this ensures availability around the clock without the need to have a computer running at home all day long. Lastly, this makes sharing the tool more comfortable, as the network configuration is standardized within the Oracle Cloud, and the IP address of the web server does not change.

If you want to run everything in the Oracle Cloud on free tier instances, the things to take into account are explained in section A.2.

A.1 Deployment

Note: If you want to execute the Node.js Web application as well as the Docker container on the Oracle Cloud Free Tier, perform the steps in section A.2 first. The additional steps which must be taken during the installation are marked with **CLOUD** in the instructions. If you are setting the project up on a local machine, you can ignore the **CLOUD** annotations.

A.1.1 Accessing the Program Code

The Code of Simple MLE Deployer is stored on GitHub[1]. To access it, perform the following steps (on Linux Mint 19.2/Ubuntu 18.04):

1. Open a console and install git by running:

```
sudo apt install git
```

2. Create a directory where you want to store (and later run the application from) with:

```
mkdir <foldername>
```

3. Go into that folder by typing:

```
cd <foldername>
```

4. Clone the git repository by running:

```
git clone https://github.com/cafntown/BA-Oracle_MLE.git
```

A.1.2 Deployment of the MLE Docker image

This section explains the deployment of the Docker image of the Oracle SQL Database with MLE. It is available for download on the Oracle Technology Network[2] in the form of a compressed tarball image. MLE stands for Multilingual Engine and enables the execution of JavaScript directly inside the database. Docker is a virtualization environment that enables users to run a process inside a container, which provides the environment with all dependencies this process needs independently of the environment the container is running in.

While Docker aims to provide complete isolation, this is not always the case in practice, and deploying MLE is one such case where the isolation does not fully work. Therefore, we highly recommend using Oracle Linux 7.7 for running this image.

Prerequisites:

- Oracle Linux 7.7
- The Docker image of Oracle MLE
- Access to all files of the git repository of Simple MLE Deployer

[1] https://github.com/cafntown/BA-Oracle_MLE

[2] https://www.oracle.com/technetwork/database/multilingual-engine/overview/index.html

A.1.2.1 installation of the Docker environment

We install the docker as explained in the Oracle Linux Documentation on Docker[3]:

1. Inside a terminal run the following steps to add the required package sources and remove an eventual old docker installation:

```
sudo -s
yum-config-manager --enable ol7_addons
systemctl stop docker
yum remove docker
```

2. We execute to install the docker engine, start it up and finally check if it is running:

```
sudo -s
yum install docker-engine docker-cli
systemctl enable --now docker
systemctl status docker
exit
```

3. To avoid having to execute all docker commands as superuser, we create a docker usergroup and add the own user to it, which then has the privilege to run docker commands.

```
sudo groupadd docker
sudo usermod -aG docker $(whoami)
```

   After those steps, a logout and login to the machine is required for the changes to take effect.

4. To check, if docker was installed right, we run:

```
docker run hello-world
```

   This should give output that can be seen in Listing 14:

[3] https://docs.oracle.com/en/operating-systems/oracle-linux/docker/docker-install-repos.html

```
1     Unable to find image 'hello-world:latest' locally
2     latest: Pulling from library/hello-world
3     0e03bdcc26d7: Pull complete
4     Digest:
sha256:49a1c8800c94df04e9658809b006fd8a686cab8028d33cfba2cc049724254202
5     Status: Downloaded newer image for hello-world:latest
6
7     Hello from Docker!
8     This message shows that your installation appears to be working
correctly.
9
10    To generate this message, Docker took the following steps:
11     1. The Docker client contacted the Docker daemon.
12     2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
13        (amd64)
14     3. The Docker daemon created a new container from that image which runs
the
15        executable that produces the output you are currently reading.
16     4. The Docker daemon streamed that output to the Docker client, which
sent it
17        to your terminal.
18
19    To try something more ambitious, you can run an Ubuntu container with:
20     $ docker run -it ubuntu bash
21
22    Share images, automate workflows, and more with a free Docker ID:
23     https://hub.docker.com/
24
25    For more examples and ideas, visit:
26     https://docs.docker.com/get-started/
```

Listing 14: Expected output of the `docker run hello-world` command

A.1.2.2 Setting up and running the Docker image

1. We create a directory in which we want to store the data (i.e. the database) of the docker image. With the command:

```
    mkdir <foldername>
```

2. Next we change the working directory to this folder with:

```
    cd <foldername>
```

3. Inside this folder we create two folders named JS and DB running the command:

```
    mkdir DB && mkdir JS
```

4. Then we change the permissions of the DB folder so that the docker image will not have an issue writing to it by running:

```
    chmod 777 DB
```

   CLOUD

   Now we copy the docker-mle tarball image to the machine. If we use a remote machine with ssh to run this, we disconnect, use scp to copy the file, and then reconnect again to the remote machine like so. The first argument in the scp command is the path to where the image is stored on your machine, and the second argument is the path where you want to store the image on the remote machine. For ease of use, take the folder `<foldername>` you created three steps before.

```
    logout # to disconnect from the remote machine
    scp path/to/tarball/mle-docker-0.3.0.tar.gz
 user@machine:path/to/remote/image/
    ssh username@publicIPAdress # reconnect to the remote machine
```

5. Once this is done, we change directory into the folder where the tarball is located using `cd` and execute to load the image into docker:

```
    docker load --input path/to/tarball/mle-docker-0.3.0.tar.gz
```

6. After that, we create the container with the following code. Creating the container will take some time until it clearly states if the database was set up successfully. During our deployment this took roughly 5-10 minutes.

```
    docker run --name oracle_mle \
   -p 1521:1521 -p 5500:5500 \
   -e ORACLE_SID=ORCLCDB \
   -e ORACLE_PDB=ORCLPDB1 \
   -e ORACLE_PWD=tiger12345 \
   -v /home/username/foldername/DB:/opt/oracle/oradata \
   -v /home/username/foldername/JS:/home/oracle/myproject \
   mle-docker:0.3.0
```

7. **CLOUD**

   After the setup is finished you can disconnect from the remote machine with

```
logout
```

8. We navigate to the folder `/JS/scripts` inside our local copy of the Git repository of Simple MLE Deployer using `cd` .

9. Now we can connect to the database and run the setup script with the following command:

```
sqlplus scott/tiger@localhost/ORCLPDB1 @setup.sql
```

   CLOUD

   Replace the localhost with the IP Address of your cloud instance.

10. We can stop, restart or start the docker container with the commands:

```
docker stop oracle_mle
docker restart oracle_mle
docker start oracle_mle
```

A.1.2.3 Personal Notes

In theory, Docker decouples the software environment of the host from the environment inside the container. This seems to be valid to some extent, but not completely.

In the process of trying to deploy this image, I tried various Linux Distributions. Probably by sheer luck, I managed to get it running on my machine, which at the time was running Linux Mint Cinnamon 19.2. Later I tried with Ubuntu Server 16.04 as well as 18.04 and an unknown version of CentOS, all with no luck. Only moving to Oracle Linux 7.7 allowed spinning up that docker image in the end.

So if you want to make sure you are not running into issues, take the installation steps on Oracle Linux! We are unsure why exactly this is, but we assume that it has something to do with the underlying differences in the Linux kernels of different distributions since Docker does not virtualize the kernel.

A.1.3 Deployment of the Simple MLE Deployer application

This section provides information on how to deploy the Node.js application on Linux. They have been verified to work on Ubuntu Server 18.04 and Linux Mint 19.2. Node.js provides a JavaScript runtime without requiring a browser. This enables developers to develop applications with other targets than modern browsers.

A.1.3.1 Prepare the Machine for Deployment

Installing the Oracle Instant Client

1. Next we download the files of the Oracle Instant Client[4]. These files are required by the OracleDB npm package. Download the following rpm files:
   - Basic Package (basic)
   - SQL*Plus Package (sqlplus)
   - SDK Package (devel)
2. [5]Install alien with:

```
sudo apt install alien
```

   This tool converts rpm packages to Debian packages and installs them.

3. We install libaio1 with the next command. It is a dependency of the Oracle Instant Client.

```
sudo apt install libaio1
```

4. We change directory into the folder where we saved the (rpm) installation files by using `cd`. And install them with the following commands:

```
sudo alien -i oracle-instantclient*-basic*.rpm
sudo alien -i oracle-instantclient*-sqlplus*.rpm
sudo alien -i oracle-instantclient*-devel*.rpm
```

5. Next we open the file `/etc/ld.so.conf.d/oracle.conf` with our editor of choice. For example with:

```
sudo vim /etc/ld.so.conf.d/oracle.conf
```

6. And add the following line to the file. At the time of writing `<your version>` was replaced with 19.6.

```
/usr/lib/oracle/<your version>/client64/lib/
```

   If we are running a 32-bit system, we add the following line to the file instead:

```
/usr/lib/oracle/<your version>/client/lib/
```

7. We reload the config by running:

```
sudo ldconfig
```

[4]https://www.oracle.com/database/technologies/instant-client/downloads.html
[5]Source: https://askubuntu.com/questions/159939/how-to-install-sqlplus#207145

Installing Node.js

1. If not already installed, we install git on the machine we want to run the Node.js application by running:

```
    sudo apt install git
```

2. Next up we get the Node version manager (nvm) by running:

```
    curl -o-
 https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

3. Logout and login to your machine for the changes to take effect.

4. And install the latest version of Node.js by typing:

```
    nvm install node
```

CLOUD

[6]Once connected via SSH, we execute the following commands on the machine to grant all connections to and from the outside. We are aware of the fact that this, in general, is a terrible security practice. However, configuring the `iptables` of a Linux is ultimately outside of the scope of this thesis.

```
    sudo -s
    iptables -P INPUT ACCEPT
    iptables -P OUTPUT ACCEPT
    iptables -P FORWARD ACCEPT
    iptables -F
```

A.1.3.2 Deploying the Node.js Application

1. Get the source Code as described in the subsection A.1.1

2. Once the code is completely downloaded, we use

```
    cd BA-Oracle_MLE/JS/
```

   to move into the JS folder of the git repository. There we execute

```
    npm install
```

   to install the required dependencies for the application to run.

3. Next we move into the “WebApp” folder by using

```
    cd WebApp/
```

[6]Source: https://stackoverflow.com/a/54810101

again and run:

```
node webapp.js
```

If we want to run the application disconnected from the shell instance, we can do so by running[7]

```
nohup node webapp.js  > /dev/null 2>&1 &
```

instead of `node webapp.js`.

A.1.3.3 Personal thoughts

In the process of the programming work for this project, I learned about the node version manager[8] (nvm). It tremendously helps to avoid rabbit holes and pitfalls in the process of installing and working with node.

A.2 Additional steps for deploying it on Oracle Cloud free tier instances

Deploying on a server in the cloud provides benefits, like a fixed IP address that makes sharing the tool easier. Additionally, there is no need to have a computer running all day long at home.

Before running the application on Oracle Cloud instances, a few additional steps are required. After having completed those, we can follow the instructions above.

1. We setup our Oracle Cloud account
2. If we have not done that already, generate ssh keys on the machine from which we want to connect (i.e., setup) to the instances. To do that we execute:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

3. In the oracle cloud console, we click on the hamburger menu in the top left ⇒ compute ⇒ instances. Then a click on the button “Create Instance ”. See Figure A.1 for precise instructions where to click.
4. In the name field, we enter the desired name.
5. There, the preferred image can be selected. **If we want to run the docker image, we must choose the Oracle Linux 7.7 as image source**.
6. Next, we copy and paste the contents of the file at `~/.ssh/id\_rsa.pub` into the field in the cloud console. See Figure A.2 for an example configuration.

[7] Source: https://stackoverflow.com/questions/29142/getting-ssh-to-execute-a-command-in-the-background-on-target-machine

[8] https://github.com/nvm-sh/nvm#installing-and-updating

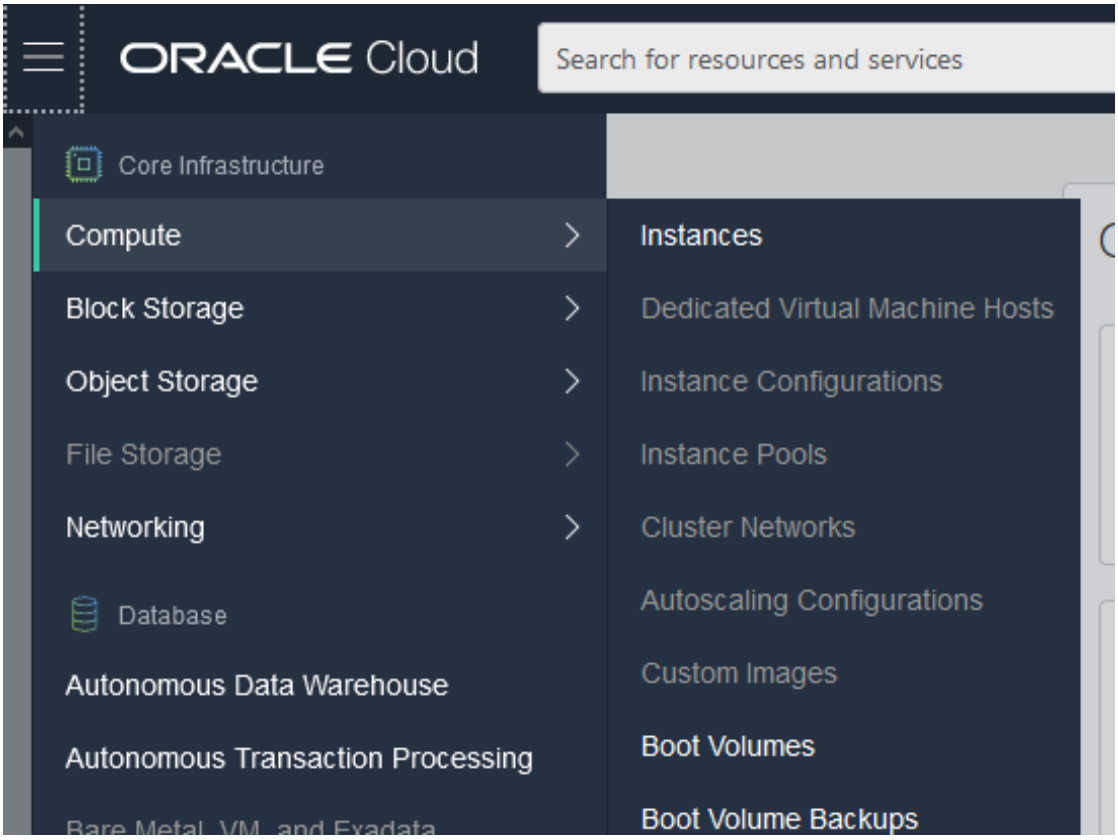

Figure A.1: User interface of Oracle Cloud showing the path to creating an instance

7. During setup, I happened to uncheck the two Oracle Cloud Agent (advanced) options in order to avoid any unwanted influence. However, it should not matter whether those are enabled or not.

8. Then we click on the button “Create” and wait for some minutes until the instance is running.

9. Back in the Oracle Cloud Console, we click on the “Virtual Cloud Network” of the machine ⇒ “Security Lists” on the right side ⇒ Click on the only entry in the list, named something like “Default Security List...” ⇒ Click “Add Ingress Rules”

10. As all free tier instances are part of the same virtual network, we must add all the required rules here. This means we need to add three rules with the following settings:

    - Stateless: unchecked
    - Source Type: CIDR
    - Source CIDR: 0.0.0.0/0
    - IP Protocol: TCP

- Source Port Range: leave empty
- Destination Port Range: for the first rule, insert 3000 here. For the others 1521 and 5500.
- The description is optional and can be left empty
- Click “Add Ingress Rules”

11. When the Public IP Address appears, we copy it and wait a bit longer. Then we connect to the instance by executing this:

    `ssh username@publicIPAdress`. Depending on the image we chose beforehand, the username can be “opc” or “ubuntu” or something else. We just try one of these, and you will get a response as to what we should connect.

We are aware that allowing all connections in a firewall is not the best practice. However, it is outside of the scope of this thesis.

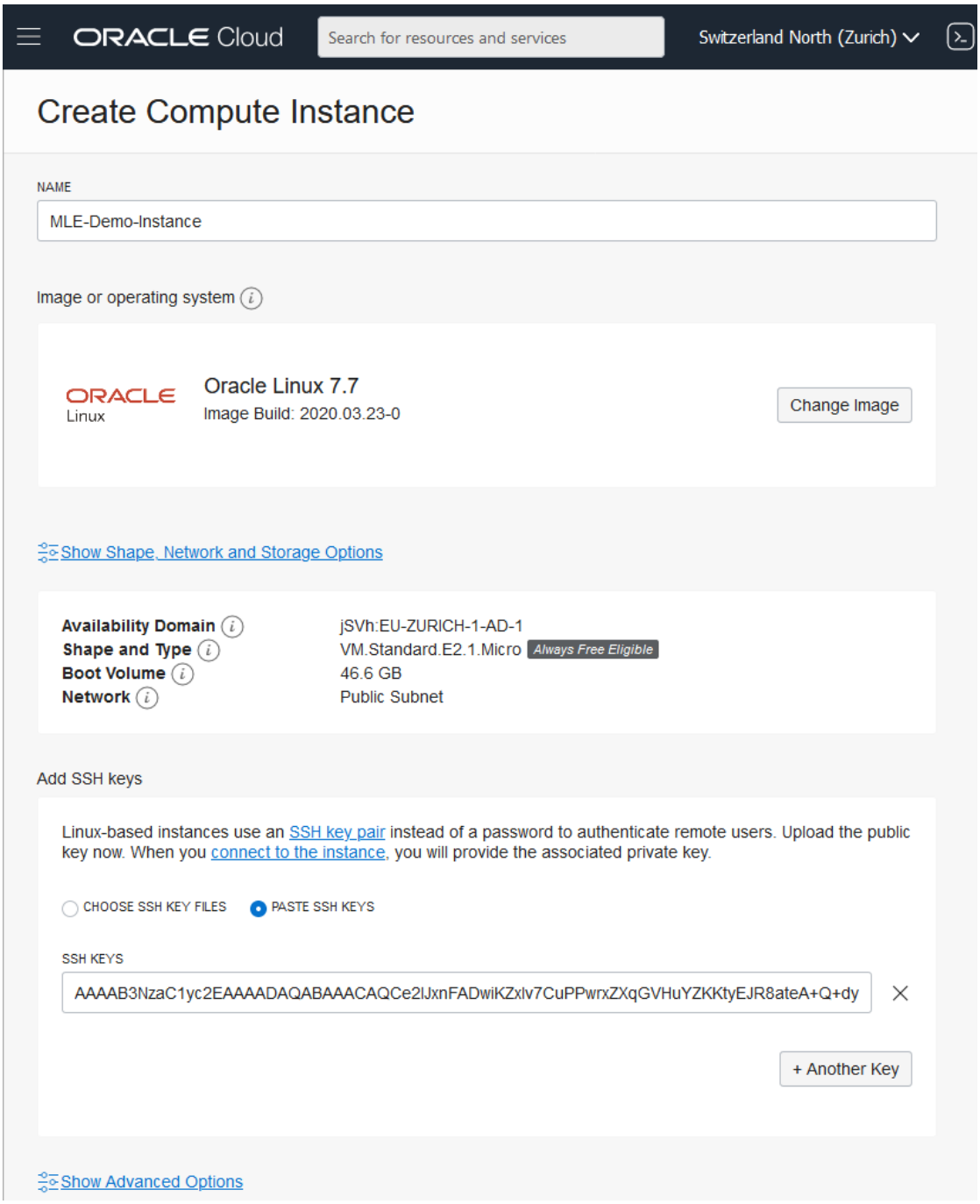

Figure A.2: Mask in Oracle Cloud for deploying a separate instance

Bibliography

[1] McLaughlin. *Oracle Database 12c PL-SQL programming*. McGraw-Hill Education, 2014.