JEEDY – ORDS MANAGEMENT API

JUNE - AUGUST 2021

AUTHOR
Marcel Ochsendorf

IT-DEP-DAR

SUPERVISORS:
Antonio Nappi
Artur Wiecek

CERN openlab

PROJECT SPECIFICATION

The CERN IT department offers the hosting of custom applications running on an Oracle-Database. These applications can be of one of the following types:

- APEX
- ORDS
- ORAWEB

Also, each application can have several configuration parameters to control its behaviour or on which database the application should run.

To manage these services in an automated way, a system called **DAD** is used, to deploy these services, regarding a setup configuration stored in a database schema.

The DAD system contains of two parts:

- DAD database – contains setup configuration
- DAD_EDIT APEX - management application
- ORDS_CONFIG - utility to spawn created application

The goal of this project is to create a further possibility to modify the DAD database entries using a custom REST API.

This should be archived though an ORDS based REST API, which should be able to perform the same tasks as the DAD_EDIT APEX application.

So, the aim of this project is:

- Create a database and ORDS environment using Docker
- Investigate the logic of the DAD_EDIT APEX application
- Recreate the behaviour as an ORDS module

ABSTRACT

The number of functionalities covered by the Oracle database increases from release to release.

Thus also the possibilities to communicate with it from outside. More and more systems use HTTP requests in the form of REST API calls to interact with other systems.

Through Oracle ORDS this communication is also possible to an Oracle database. This reduces the number of systems that need direct access to a database via a JDBC connector.

In this project the creation of an ORDS module for a database schema is implemented and documented using the open-api standard. Subsequently, an application that previously required direct JDBC access to the database will be rewritten to enable communication with the resulting ORDS API.

TABLE OF CONTENTS

1. ORGANSIATION

This report is into four parts:

- Project introduction and application overview
- ORDS REST API implementation
- API integration
- Conclusion

2. INTRODUCTION

In 2021 CERN already uses ORDS in its productive systems. As shown above users can already setup their own ORDS application on CERN database instances.

Only on the management site of these applications, ORDS is not used yet, but all requirements to implement a ORDS managements API are already satisfied for development databases such devdb19 or on production databases like cerndb1.

On the security site, CERN uses a Single-Sign-On system, and a granular permission system called e-groups to allow users access to a specific database.

These systems are already enabled inside of the database instances and permissions to a specific database schema can be setup using the roles system inside the databases ORDS module.

The uses version of Oracle database is for devdb19 and cerndb1, version 19, so not the latest but it offers all needed ORDS features for this project.

a. DAD_EDIT APEX APPLICATION OVERVIEW

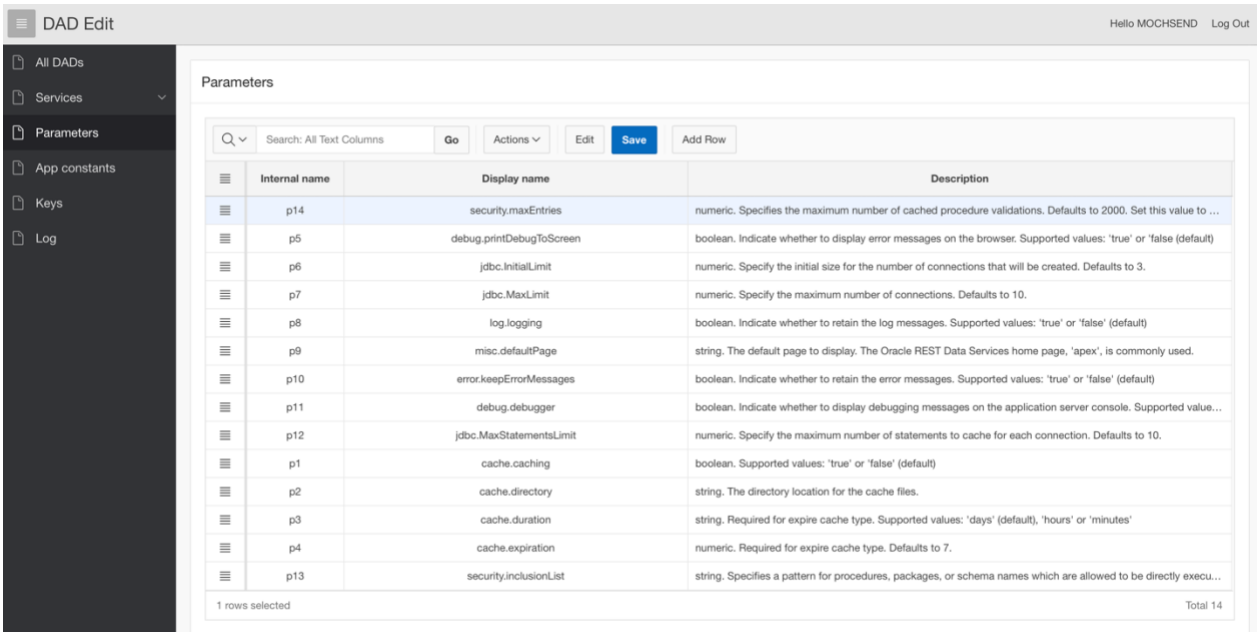


Figure 1DAD_EDIT APEX APPLICATION

The first step is to identify the basic functionality of the DAD management application. This offers the creation and modification of new services and allows the parameterization of each service.

The tool is an APEX (Oracle Application Express) application, which runs directly on the Oracle database and the Tomcat server. The development of such application is very easy, by the IDE provided by the APEX software.

The IDE uses a simple drag and drop system to create such application directly online in a web browser and offers features like, search-bar and row delete dialogs without coding.

b. Function summary of the REST API

After some research in the DAD management application the API should offer the following functionalities:

i. CREATE, MODIFY, DELETE, LIST of the following database tables

- APP_CONFIGURATION
- APP_CONSTANT
- DAD
- DAD_TYPE
- LOGINFORMATION
- RELATION_BETWEEN_COLUMN_NAMES (SCHEMA)
- SERVICE
- SERVICE_ORDSVERSION
- SERVICE_TYPE

ii. USAGE OF STANDART REST METHODS

- **GET**, list entries
- **POST**, modifying entries
- **PUT**, create new entries
- **DELETE**, delete entry

iii. RESPONSE

In general, all responses should use status codes (404, 200), and the JSON format as the response media type. An additional error-message is also nice to have in the JSON response.

LIST ENTRIES FILTER OPTIONS

Most of the list actions should offer additional optional filter parameters, to allow the reduction of the resultset. For example, filter by a specified ID field. The available filter options depend on the type of schema. Some examples are listed below:

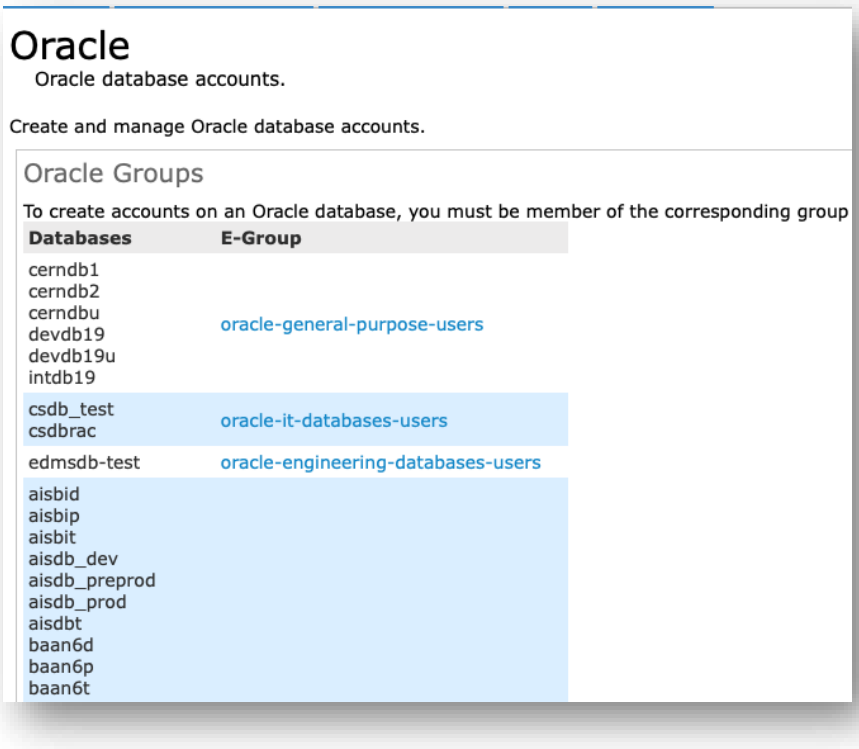
- ID
- NAME
- TYPE
- LIMIT RESULT COUNT
- ORDER ASC/DSC

3. ORDS REST API IMPLEMENTATION

a. DEVELOPMENT SYSTEM SETUP

To start with the development of the ORDS API, a test database with ORDS functionality is necessary. CERN offers different resources for every employee. Starting from simple access to VMs or cloud storage. There is also an option or Oracle databases for several different purposes and versions.

DEVDB19 was used as test and development database. The DAD database is running on CERNDB1 which is the productive database for the system. After tests and verifications, the ORDS API is deployed on CERNDB1.



Oracle

Oracle database accounts.

Create and manage Oracle database accounts.

Oracle Groups

To create accounts on an Oracle database, you must be member of the corresponding group

Databases	E-Group
cerndb1 cerndb2 cerndbu devdb19 devdb19u intdb19	oracle-general-purpose-users
csdb_test csdbrac	oracle-it-databases-users
edmsdb-test	oracle-engineering-databases-users
aisbid aisbip aisbit aisdb_dev aisdb_preprod aisdb_prod aisdbt baan6d baan6p baan6t	

ORDS MODULE

The general structure of a ORDS [1] system consist of the following objects:

- Module
- Template
- Handler

Each module can contain several templates and every template can contain a handler for each HTTP method (get, post, put and delete).

The basic URL schema of a ORDS module:

http://devdb19/ords/<SCHEMA>/<MODULE>/<TEMPLATE>

In case of the ORDS management API, the database schema is called DAD_EDIT3. The name of the ORDS module is called **ords_mgmt_api**. The module is the "folder" contains all routes for the different tables.

So, for each table a template was created, which is named exactly like the table.

- **http://devdb19/ords/dad_edit3/ords_mgmt_api/dad**
- **http://devdb19/ords/dad_edit3/ords_mgmt_api/service**
- **http://devdb19/ords/dad_edit3/ords_mgmt_api/schema**

In every template a handler was defined for each action [2], which should be performed on the table. A handler contains the HTTP method, query parameters and can execute/perform different actions on a database table. In this case each handler executes written PL/SQL code on the database and calls PL/SQL procedures.

```
ORDS.DEFINE_TEMPLATE(
    p_module_name    => 'ords_rest_mgmt_api_v1.0',
    p_pattern        => 'appconfiguration',
    p_priority       => 0,
    p_etag_type      => 'HASH',
    p_etag_query     => NULL,
    p_comments       => NULL);
ORDS.DEFINE_HANDLER(
    p_module_name    => 'ords_rest_mgmt_api_v1.0',
    p_pattern        => 'appconfiguration',
    p_method         => 'GET',
    p_source_type    => 'plsql/block',
    p_items_per_page =>  25,
    p_mimes_allowed  => '',
    p_comments       => NULL,
    p_source         =>
'BEGIN
   get_appconfiguration_prd(
      i_key => :i_key,
      i_value => :i_value,
      i_limit => :i_limit,
      o_result => :o_result,
      o_error => :o_error
   );
END;'
    );
ORDS.DEFINE_PARAMETER(
    p_module_name        => 'ords_rest_mgmt_api_v1.0',
    p_pattern            => 'appconfiguration',
    p_method             => 'GET',
    p_name               => 'error',
    p_bind_variable_name => 'o_error',
    p_source_type        => 'RESPONSE',
    p_param_type         => 'STRING',
    p_access_method      => 'OUT',
    p_comments           => 'o_result');
```

Figure 2 ORDS HANDLER DEFINITION [2]

b. PL/SQL PROCEDURES | FUNCTIONS

After a handler is called, a defined PL/SQL procedure is executed and the resultset or error is passed back to the handler to send the response to the client or web-browser. The procedure contains the SQL query, which is executed on the database. For example, the select all rows of a table:

SELECT * FROM DAD WHERE 1=1 AND DAD.ID = 42

It is also possible to insert the in the handler defined parameters into the SQL statement

SELECT DAD.ID FROM DAD WHERE 1=1 AND DAD.ACTIVE=:<PARAMETER>

For simple SQL select queries this method is perfect and fits the most cases. To implement complex filtering, inserting, or modifying of rows, **dynamic SQL**[3] was used to perform this task.

Here the query string is assembled depending on the given parameters dynamically and at the last step, the query string is executed [4]. The resultset and error of the execution will be returned the same way as with the SQL/PL Procedure to the ORDS handler.

```
  CREATE OR REPLACE EDITIONABLE FUNCTION "GET_SCHEMA_FKT" (

    I_INCLUDE_DAD IN INTEGER, -- INCLUDE DAD ENTRIES
    I_INCLUDE_DAD_DETAILED IN INTEGER, -- INCLUDE ALL DAD ENTRIES

    I_ID IN SCHEMA_TABLE.ID%TYPE, -- SELECT SERVICE WITH ID
    I_NAME IN SCHEMA_TABLE.NAME%TYPE, -- SEARCH SERVICE NAME
    I_DADID IN SCHEMA_TABLE.DAD_ID%TYPE, -- SEARCH CLUSTER_NAME
    I_ISENCRYPTED IN SCHEMA_TABLE.IS_ENCRYPTED%TYPE, -- SEARCH ENTITY
    I_PASSWORD IN SCHEMA_TABLE.PASSWORD%TYPE -- SEARCH FRONTEND_ENITY

)
RETURN SYS_REFCURSOR
AS
    L_QUERY     VARCHAR2(32767);
    VAR_REF SYS_REFCURSOR;
BEGIN

    --BASIC SELECT
    L_QUERY := 'SELECT SCHEMA_TABLE.ID, SCHEMA_TABLE.NAME, SCHEMA_TABLE


    -- INCLUDE DAD ENTRIES
    IF I_INCLUDE_DAD = 1 then
        L_QUERY := L_QUERY || ', DAD.ID AS DAD_ID, DAD.NAME AS DAD_NAME
        -- INCLUDE ALL DAD COLUMNS
        IF I_INCLUDE_DAD_DETAILED = 1 THEN
        L_QUERY := L_QUERY || ', DAD.P1 AS DAD_P1, DAD.P2 AS DAD_P2, DA
    END IF;
    END IF;



    -- ADD TABLE
   L_QUERY := L_QUERY || ' FROM SCHEMA_TABLE';
```

Figure 3 PL/SQL FUNCTION [4]

c. REST API ROUTE TESTING | DOCUMENATION

During and after implementing the different ORDS routes, each API call is tested using an application used **Postman**. This allows to save and repeat different HTTP request and allows easy management of query parameters, including simple documentation for their purposes.

The result of a performed HTTP is directly shown to the user, including header and cookie information. Postman[5] can display the body content of the response directly formatted into the media type of the response [6]. JSON responses will directly “beautified” into a nicely readable format.

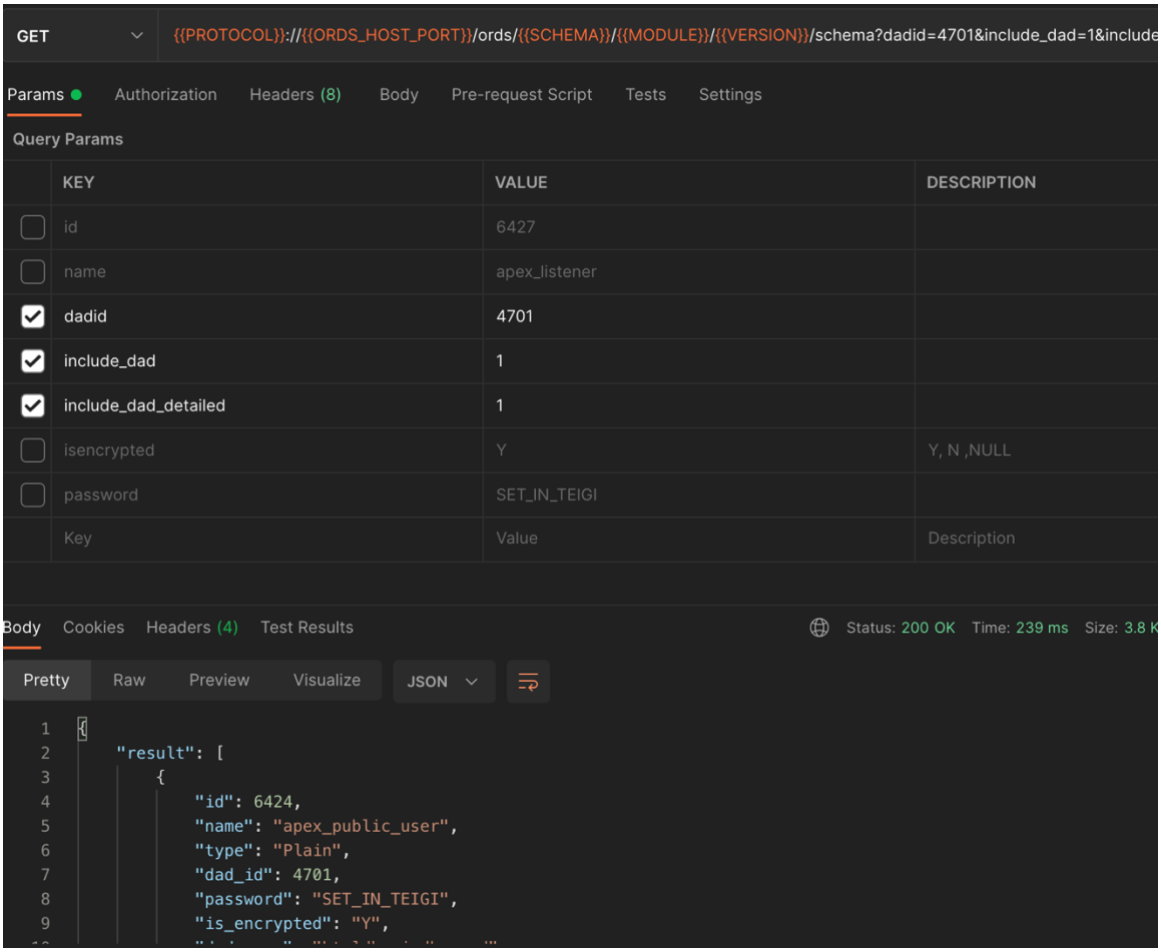


Figure 4 POSTMAN REQUEST AND JSON RESPONSE [6]

Another advantage is collection feature of Postman. Several different requests can be grouped together into folders, to keep the overview over the implemented routes.

After finishing all routes, the export feature was used to export the collection into an open-api format. Because Postman is a stand-alone tool and has to be installed on a system to view or run requests, it is not the ideal tool for an API documentation shared across the web. Each time the exported collection need to be imported manually. To resolve this issue another tool called **Swagger**[7] was used, which allow a browser based representation of the exported collection. It is also directly integrated into Gitlab. It offers basically the same features of Postman (except the testing), run requests and show the response but over a web-browser[8]. Also authentication can be setup using basic-auth, OAuth and several other methods.

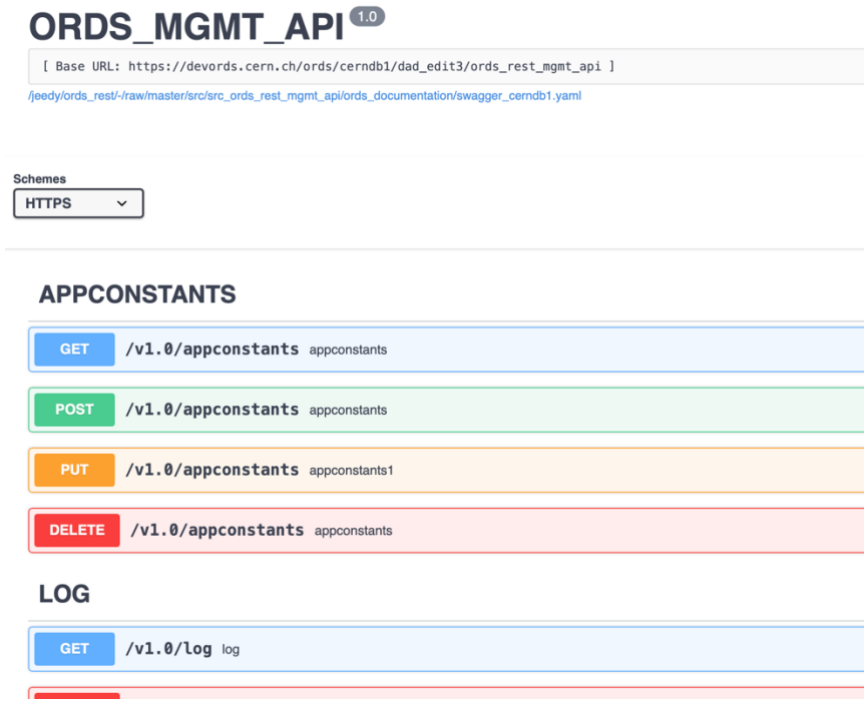


Figure 5 SWAGGER COLLECTION [8]

4. REST API INTEGRATION

After completion of the API and after thorough testing, it should be used productively. One advantage of the newly created API is that direct database access is no longer necessary. So, no need to directly share username and password of the database schema in an application and due to the modular permission-system which can be setup for each ORDS Module, it is also possible the give users only read access to selected routes.

On application which uses the direct database access, is the ORDS_CONFIG_IMAGE application, which spawns the in the DAD database listed services. This application needs access to the database to collect all necessary information about the services and their parameters to setup these on the target system.

The default way is that at startup, the application login to the database server and executed several SQL queries on the schema [9].

```
def get_dads(connection, service_name, active=True):
    logger.info('Retrieving all DADs of the service %s with state active==%s' % (service_name, active))
    service = get_service(connection, service_name)
    if not service:
        logger.error("A service called %s could not be retrieved." % service_name)
        return []
    cur = connection.cursor()
    cur.prepare('select * from Dad where service_id = :s_id and active = :active order by id')
    cur.execute(None, {'s_id': service['id'], 'active': 'Y' if active else 'N'})
    res = cur.fetchall()
    dads = []
    for r in res:
        dads.append(dict(zip(tuple(cd[0].lower() for cd in cur.description), r)))
    cur.close()
    logger.info('DADs retrieved')
    return dads
```

Figure 6 ORDS_CONFIG_IMAGE DATABASE ACCESS [9]

If there is any changes in the database schema, this application needs to be modified. By using the new ORDS REST API this problem can be ignored, as long as the JSON response of the API didn't changed.

Also the application is only working with Oracle Databases as data source and the Oracle client libraries have to be installed on the system. So as the first productive usage of the new API, this application was rewritten in order to use the API as its data source [10].

The main functions, to access the database was written in Python. To make HTTP requests working the Python36-Requests package was used and the necessary functions were rewritten. In general the functions are a bit longer, due to more error handling on the HTTP requests.

```
def get_dads(service_name, active=True):
    logger.info('Retrieving all DADs of the service %s with state active==%s' % (service_name, active))
    service = get_service(service_name)
    if not service:
        logger.error("A service called %s could not be retrieved." % service_name)
        return []

    logger.info('Retrieving service %s' % service_name)
    query = {
        'detailed': 1,
        'serviceid': service['id']
    }
    if active:
        query['active'] = 'Y'
    else:
        query['active'] = 'N'

    dads = []
    req = requests.get(dadEdit3_config.dadEdit3_ords_base_url + "/dad", params=query, auth=dadEdit3_utils.get_ords_auth())

    if req.status_code >= 200 and req.status_code < 300:
        req_json = None
        try:
            req_json = json.loads(req.text)
        except ValueError:
            logger.error("json.loads parse error")
            req_json = None

        if req_json is not None and 'result' in req_json:
            result_list = req_json['result']
            for r in result_list:
                dads.append(r)
            logger.info('DADs retrieved')
        else:
            print("no result field in response")
    else:
        print("req.status_code not in 200 range")
        dads = []
```

Figure 7ORDS_CONFIG_IMAGE API ACCESS [10]

5. CONCLUSION

The goal of implementing an ORDS REST API was a success. After some time to get into the software, setup the dev system and get warm with the Oracle Database system, the implementation of the required functionalities was a straightforward procedure.

The time spent studying the DAD_EDIT APEX application to understand the system in depth helped me to plan my work in a structured way.

The final implementation of the ORDS API could thus be implemented well. The final deployment of this on a production system completed the project and allows to build on it in the future.

The project continues to offer the potential to be further developed. Here, for example, it is possible to realize more functionality in the API, which is otherwise carried out via other applications via a direct database access.

Also, the creation of client libraries for different programming languages is no problem thanks to the Swagger documentation. Thus, other systems that want to use the API can also integrate it with ease.

6. ACKNOWLEDGMENT

I would like to say a big thank you to my supervisors **Antonio Nappi**, **Artur Wiecek** and **Luis Rodriguez Fernandez**, who supported me with their enormous knowledge and help in this project and thus contributed to the successful completion of this project. I would also like to thank all IT-DEP-DAR colleagues for the many interesting discussions and insights I was able to gain.

A special thanks also goes to all who participated in the Summer-Student and **openlab** program, which made this student program a very special one.

7. REFERNCES

[1] https://docs.oracle.com/en/database/oracle/oracle-rest-data-services/

[2] https://gitlab.cern.ch/jeedy/ords_rest/-/blob/master/src/src_ords_rest_mgmt_api/plsql_functions_procedures_ords/ords_ordsmodule.sql

[3] https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/dynamic.htm#LNPLS011

[4] https://gitlab.cern.ch/jeedy/ords_rest/-/blob/master/src/src_ords_rest_mgmt_api/plsql_functions_procedures_ords/ords_functions.sql

[5] https://www.postman.com/api-platform/

[6] https://gitlab.cern.ch/jeedy/ords_rest/-/blob/master/src/src_ords_rest_mgmt_api/ords_documentation/postman.json

[7] https://swagger.io/tools/swagger-editor/

[8] https://gitlab.cern.ch/jeedy/ords_rest/-/blob/master/src/src_ords_rest_mgmt_api/ords_documentation/swagger_cerndb1.yaml

[9] https://gitlab.cern.ch/jeedy/utils/ords-config-image/-/blob/ordsmgmtapi/dadEdit/bin/database_queries.py

[10] https://gitlab.cern.ch/jeedy/utils/ords-config-image/-/blob/ordsmgmtapi/dadEdit/bin/database_queries_ords.py