# NPRG045 Ročníkový projekt (Individual project)

Štěpán Klos

In this document, we will guide the reader through the process of setting up a custom Kubernetes cluster. We will showcase its specific features, aiming to facilitate the deployment and execution of web applications within the cluster.

Kubernetes is an open-source container orchestration tool bringing new modern approach for running web applications in the cloud. Kubernetes was created by Google in 2014 and written in Go. Running production on open-source software, however, comes without any guarantees. Thus, a market for commecrial products arised. In this document we will not only present the setup per-se but also compare managed providers with self-hosted commercial solutions to self-hosted plain Kubernetes.

We will break down each step of the process ranging from overview to actual deployment on specific cloud provider.

## 1 Problem description

There are several motivations that led to the creation of Kubernetes. Environment isolation and scaling are the most significant ones. We want to have an "universal runtime environment"of various technologies to satisfy needs of developers and necesities of our company. These might range from developers needing various versions of PHP and different installed libraries to robust .NET backend, Remote Dictionary Server for session sharing, different frontends with client-side routing defined by developer (with no need of editing webserver or breaching rules for htaccess) or even microservices.

### 1.1 Stack mismatch across projects

We can encouter different needs within organization. Oldschool approach to having servers with usually one runtime fails since each user can have slightly different needs.

### 1.1.1 Backend tech and library needs

We have more (not necessarily) PHP backend projects. We need isolation to accomodate different backend technologies and version. There even might a be demand for different versions of one specific library.

- Applications running PHP 5.6.0 (ft. MySQL).
- Applications running PHP 8.0.0 (ft. MySQL).
- Application running PHP 7.2.24 + imagecreate + ZipArchive (ft. MySQL).
- Applications running Node.js + custom stuff from package.json.

### 1.1.2 Frontend configuration stuff

Configuration might be tricky for different reason. **Firstly**, we need to have **rewrites** allowed.

```
a2enmod rewrite
systemctl restart apache2
```

**Secondly**, client-side routing settings of the webserver might not be overridable via **.htaccess**.

```
Options +FollowSymLinks
Options -MultiViews
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_URI} !^/route-exception/
RewriteRule ^ index.html [QSA,L]
```

The settings has to be done on root of Apache which is a request (and potential security breach) on the administrator.

```
# :443 for https
<VirtualHost *:80>
  DocumentRoot /var/www/html
  <Directory /var/www/html/>
      AllowOverride All
      Options FollowSymLinks
  </Directory>
  <IfModule mod_security2.c>
      SecRuleEngine DirectionOnly
  </IfModule>
</VirtualHost>
```

## 1.2 Scaling

Scaling multiple applications individually or hosting them on a single server can lead to scalability challenges. However, Kubernetes provides a solution by enabling communication with a cluster consisting of independent worker nodes that can be attached. This allows workloads to scale automatically, making it an excellent choice for handling variable production demands.

### 1.3 Different images preparation

From bottom linux, to other layers as php/alpine.

## 2 Technical construction of Docker images

There are 3 key components for Container Isolation in Docker and standards and organizations making it possible to exist. In summary, chroot, namespaces, and cgroups play crucial roles in Docker. Chroot provides a separate root file system for containers, enhancing security. Namespaces isolate processes and resources, preventing interference between containers and the host system. Finally, cgroups enable Docker to manage and allocate system resources efficiently, ensuring fair usage among containers and maintaining system stability.

### Docker internals - chroot

In Docker, chroot (change root) is used to create an isolated environment for running processes. It allows Docker to create a root directory that is different from the host system's root directory, effectively limiting the view of the file system for the processes running within the container. This helps to enhance security by isolating the container's file system and preventing access to sensitive files and directories on the host system. Chroot provides a lightweight form of isolation by limiting the visibility of the container's file system, but it does not provide other forms of isolation such as process or network isolation.

### Docker internals - Namespaces

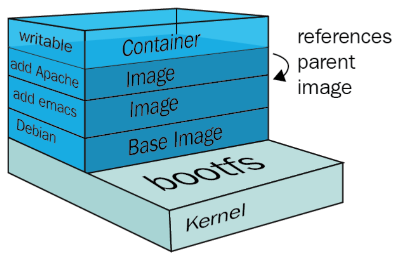
Namespaces are another key feature used by Docker to provide process isolation. Docker leverages various namespaces such as PID (process ID), mount, network, and more. Each namespace creates a separate instance of a particular resource for each container, effectively isolating the containers from each other and from the host system. For example, the PID namespace ensures that processes running inside a container only see their own processes and not processes from other containers or the host system.

### 2.1 Docker internals - cgroups

Control groups **cgroups** play a major role in isolation of running processes within Linux kernel.

### 2.2 Docker Image Structure

Images are layered... built...etc.



## 2.3 OCI (Open Container Initiative)

The Open Container Initiative is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes. It bridges gaps between Docker and other container formats and connects developers who are interested in dockerization in very friendly (non competitive) manner.

## 2.4 History

Docker was not first

## 2.5 Breakdown of Kubernetes objects

Kubernetes clusters have similar architecture regardless of provider or author. Despite some differences, the main core objects and flows are the same. We will

### Cluster Architecture

Cluster architecture refers to the arrangement and organization of nodes within a Kubernetes cluster. In a Kubernetes cluster, there are different types of nodes, including the master node and worker nodes. The master node manages the overall cluster, while the worker nodes execute the tasks and run the containers. The Kubernetes API is a core component that allows communication and interaction with the cluster, enabling users to control and manage the cluster's resources and workloads.

Nodes - Master Node, Worker Nodes, (The Kubernetes API)

### Workload

Workload in Kubernetes refers to the tasks or applications that run within the cluster. Kubernetes provides several resources to define and manage workloads effectively. Deployments are used to define and manage pods, which are the basic units of execution in Kubernetes. ReplicaSets, controlled by deployments, ensure the desired number of pod replicas are running. These resources enable easy scaling, rolling updates, and self-healing for applications running in the cluster.

Deployments - define Pods and ReplicaSets

#### Services, Load Balancing, and Networking

Services in Kubernetes provide a stable network endpoint to access a set of pods. They act as an abstraction layer, enabling load balancing across multiple pods and allowing seamless communication between them. Services have a specific port through which they can be accessed and a targetPort that defines the port where the pods receive traffic. Load balancing ensures that the traffic is distributed evenly among the available pods. Ingress is another resource that provides external access to services within the cluster, allowing for routing and other advanced networking capabilities.

Ingress Service (port, targetPort)

#### Storage

Storage in Kubernetes is managed through various resources. Volumes provide a way to store data within a pod and are often used for temporary or short-lived data. Persistent Volumes (PVs) are a more long-term storage solution, decoupled from pods, and can be dynamically provisioned or pre-allocated. Persistent Volume Claims (PVCs) are used by applications to request specific storage resources from PVs, allowing for flexibility and scalability in storage management.

Volume - Persistent Volume Persistent Volume Claim

#### Configuration

In Kubernetes, configuration management involves handling sensitive information and application settings. Secrets are resources used to store and manage sensitive data, such as passwords, API keys, or TLS certificates. They are encrypted and can be securely accessed by pods. Secrets are commonly used to provide secure access to external services or to configure applications with sensitive information.

Secret

### 2.5.1 Tasks

HorizontalPodAutoscaler (HPA) is a resource in Kubernetes that automates the scaling of deployments based on CPU utilization or custom metrics. It dynamically adjusts the number of replicas for a deployment to ensure efficient resource utilization and accommodate changes in workload demand. By automatically scaling up or down, the HPA helps maintain optimal performance and responsiveness of applications running in the cluster. The HPA is typically attached to a deployment, allowing it to monitor and control the scaling behavior for that specific workload.

HorizontalPodAutoscaler - attached to Deployment

Setup Load balancer, Ingress, Aplikace, Servisy

## 3 Technical setup

Our technical setup on DigitalOcean (based on prev. general setup)

Technical setup of components Viz. repo but cloud agnostic Setup, nodes, services

## 4 Environment comparison

### 4.1 Self hosted vs. Managed

Self hosted - Master, Worker (attaching), machines/VPS Managed - Just service

### 4.2 Minikube

Minikube - all-in-one

- self hosted options(Open Shift, Tanzu, plain Kubernetes) vs. managed options (AWS/Azure AKS/DigitalOcean Kubernetes) - services by provider vs self-hosted within cluster (within cloud provirers) Comparison admin screenshots, + and -s, table comparison Implementation on DigitalOcean https://github.com/KlosStepan/DOKS-tutorial

## 5 Application deployments

Combination of Ingress with services, deployments

### 5.1 Deployment of Redis

### 5.2 Deployment of MySQL into Cluster

PHP app, React app, CI/CD pipelines

## 6 Cluster Administration

### 6.1 kubectl

### 6.2 Kubernetes Dashboard

### 6.3 Lens

### 6.4 CI/CD Pipelines

Worflows and Azure Pipelines

## 7 Outtro

We succeded after several tryouts and self-hosted stuff which we described. SLA option (?)

## 8 Tools and software

We are using these parts and tools:

- Oracle 18.3 [1] on server,
- ORACLE SQL Developer 21.4.3 [2],
- node-oracledb 5.3.0 [3],
- features facilitating the development and working.

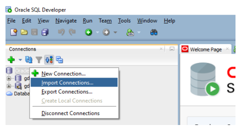
[1]https://docs.oracle.com/en/engineered-systems/oracle-database-appliance/18.3/index.html

[2]https://www.oracle.com/database/sqldeveloper/technologies/21.4.3-release-notes/

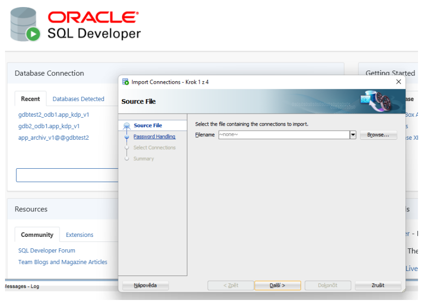
[3]https://www.npmjs.com/package/oracledb/v/5.3.0

## 9 Initial steps and setup

So we prepare ourselves by unzipping the newest version of **sqldeveloper** in our computer. Then we open it and right click **Import Connections** select encrypted json file with connector to our database provided by our collegues or us (if we created the database previously).

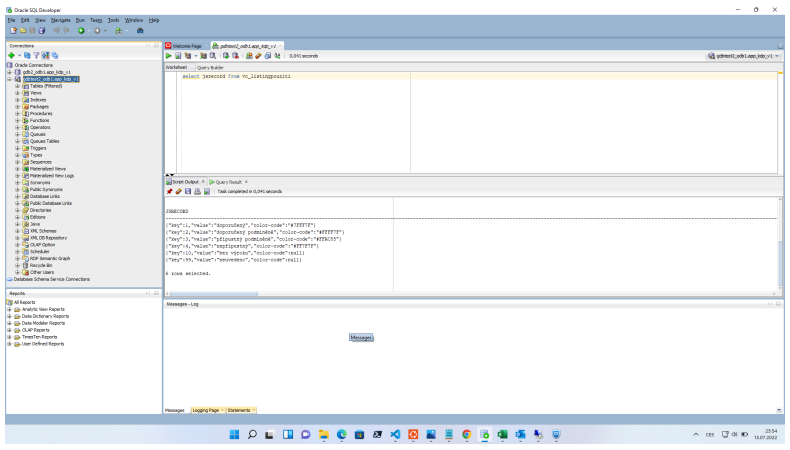


We navigate the **import connection** manager and fill out optional password to the connection file (if needed). By following this form we will have created created authorized Oracle Connections to our database(s).



Select the file containing the connections to import.

Once we are ready, we see that solution automatically comes with **production database** and **test database**. We will be working strictly on the test database until we're ready to ship tested aplication to production. After loading we can see various database related things on the left side, most importantly **Tables**, **Views**, **Procedures**, **Functions** and other stuff. We will go through these items and show some Oracle features such as BLOBs and native inserts/selects with json_object for "ORM-like"interfacing with database.



```
select jsrecord from vc_listingpouziti
```

```
JSRECORD
--------------------------------------------------------------------------------
{"key":1,"value":"doporučený","color-code":"#7FFF7F"}
{"key":2,"value":"doporučený podmíněně","color-code":"#FFFF7F"}
{"key":3,"value":"připustný podmíněně","color-code":"#FFAC85"}
{"key":4,"value":"nepřipustný","color-code":"#FF7F7F"}
{"key":10,"value":"bez výroku","color-code":null}
{"key":99,"value":"neuvedeno","color-code":null}

6 rows selected.
```

## 10 Interfacing with Oracle Database from Node.js

Oracle Database will be used for a web application. We need to connect to our database with Node.js as client language for backend. There are several things which are different from "typical PHP-MySQL connection"which is (former) de facto status quo in web development. To deliver REST API we need to:

- install **oracledb** package via npm[4],
- serialize listings in database for backend,
- send desired responses via HTTP REST endpoints.

After we install oracledb, we need to connect to database. For such task, we create object with **username**, **password** and a **connector line** which is not going to be in GIT repo for security reasons. We include use this object as an argument for connection.

**_dbCreds.js**

```
  module.exports = {
    user: 'user',
    password: 'passwd',
    connectString: "(DESCRIPTION=(ADDRESS_LIST=(FAILOVER=true)
␣␣␣␣(LOAD_BALANCE=false)(ADDRESS=(PROTOCOL=TCP)(HOST=HHHH)
␣␣␣␣(PORT=PPPP))(ADDRESS=(PROTOCOL=TCP)(HOST=HHHH)
␣␣␣␣(PORT=PPPP)))(CONNECT_DATA=(SERVICE_NAME=SSSS)))"
  }
```

**dbConn.js**

```
const oracledb = require('oracledb');
const dbCreds = require("./_dbCreds");
let database = null;
async function startDatabase() {
  database = await oracledb.getConnection(dbCreds)
}
async function getDatabase() {
  if (!database) await startDatabase();
  return database;
}
module.exports = {
  getDatabase,
  startDatabase
};
```

Noteworthy item here is the connectionString [5] which must be included to specify connection creds. Then we can execute queries thanks to classes as follows:

```
const { getDatabase } = require("./dbConn");
async function f() {
  let connection = await getDatabase();
  let query = "select␣...␣from␣view"
  result = await connection.execute(query);
}
```

[4]Run "npm install oracledb"for https://www.npmjs.com/package/oracledb
[5]For more information https://docs.oracle.com/database/121/ODPNT/featConnecting.htm#ODPNT166

## 11 Construction of REST API

We will outline several types of endpoints which are needed not only in our applicaton but in general. Some of ours, however, utilise serialization of SELECT via json_object in Oracle Database, some send picture from assembled bytestream from database BLOB, some server for insertions, etc.

### 11.1 Listing REST endpoint - GET /

Most straightforward is endpoint providing listing of elements. These endpoints return **View** (with joined lists from other table affiliated) and returns entries as JSON thanks to json_object [6]. **select * from v_listingvybaveni**

```
  CREATE OR REPLACE FORCE EDITIONABLE VIEW "APP_KDP_V1"
  ."V_LISTINGVYBAVENI" ("JSRECORD") AS
  with vp as (
    select
        vybaveni_id
        , '[' || listagg('{"rodina_kod":"' || rodina_kod
         || '", "pouzitiId":' || pouziti_id || '}',', ')
          WITHIN GROUP (ORDER BY rodina_kod) || ']' vybaveniPouziti
    from vybaveni_pouziti
    group by vybaveni_id
),
f as(
    select
        prvek_id
        , '[' || listagg('{"fotoId":'|| id ||'}',', ')
          WITHIN GROUP (ORDER BY poradi) || ']' foto
    from foto
    group by prvek_id
)
select json_object(
        'id' value v.id,
        'kod' value v.kod,
        'design' value v.design,
        'nazev' value v.nazev,
        'popis' value v.popis,
        'ergonomie' value v.ergonomie,
        'material' value v.material,
        'servis' value v.servis,
        'certifikat' value v.certifikat,
        'modifikace' value v.modifikace,
        'dostupnost' value v.dostupnost,
        'cenovahladina' value v.cenovahladina,
        'viditelnost' value v.viditelnost,
        'vybaveniPouziti' value vp.vybaveniPouziti format json,
        'foto' value f.foto format json
    )
from vybaveni v
left join vp on vp.vybaveni_id = v.id
left join f on f.prvek_id = v.id;
```

[6]More about json_object https://docs.oracle.com/en/database/oracle/oracle-database/12.2/sqlrf/JSON_OBJECT.html#GUID-1EF347AE-7FDA-4B41-AFE0-DD5A49E8B370

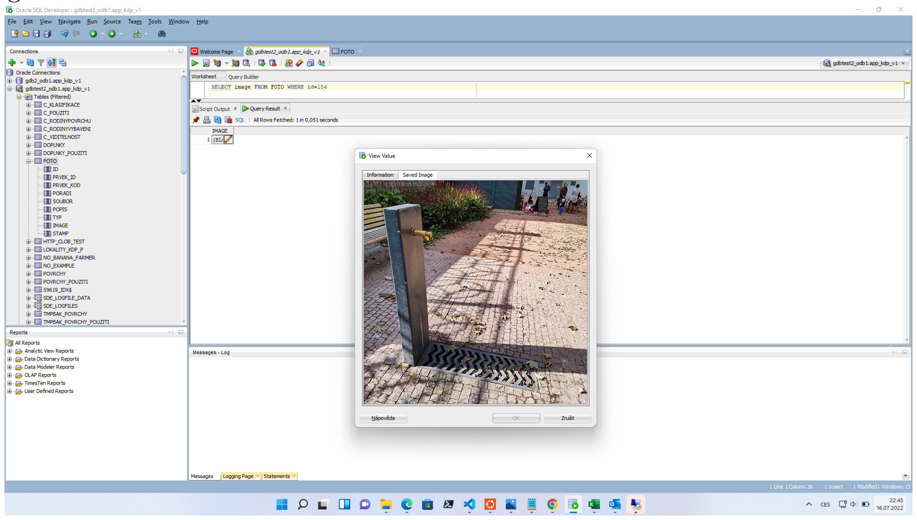
## 11.2 Image backend service via BLOB - GET /id

We store images in database in BLOBs and have endpoint which will deliver upon request. This is the function to retrieve and return BLOB [7].

```
//Foto for BE function
async function retrieveFotoById(req, res, fotoId) {
  let connection;
  try {
    connection = await getDatabase();
    const result = await connection
    .execute("SELECT image FROM FOTO WHERE id='"+fotoId+"'");
    if (result.rows.length === 0) {
      throw new Error("No data selected from table.");
    }
    const lob = result.rows[0][0];
    if (lob === null) {
      throw new Error("BLOB was NULL");
    }
    const doStream = new Promise((resolve, reject) => {
      lob.on('end', () => {
        res.end();
      });
      lob.on('close', () => {
        resolve();
      });
      lob.on('error', (err) => {
        reject(err);
      });
      res.writeHead(200, {'Content-Type': 'image/jpeg' });
      lob.pipe(res);  // write the image out
    });
    await doStream;
    connection = null;
  } catch (err) {
    console.error(err);
    await closePoolAndExit();
    } finally {
    if (connection) {
      try {
        await connection.close();
      } catch (err) {
      console.error(err);
      }
    }
  }
}
async function closePoolAndExit() {
  try {
    await oracledb.getPool().close(2);
    process.exit(0);
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
}
```

[7] https://docs.oracle.com/database/121/TDPPH/ch_twelve_blobs.htm#TDPPH183

Selected image get streamed with header to client via backend as a HTTP response. Preview of parametrized query is possible to be viewed in SQL Developer even with "image representation of bytestream"- which in this case is a JPEG image.



```
SELECT image FROM FOTO WHERE id=154
```

All Rows Fetched: 1 in 0,051 seconds

Application endpoint for image, therefore, returns an image, which in reality doesn't even have to occupy a space as a file on server's filesystem, which for dummy previews like these is very elegant way of simplifying application.

Requests, thus can arrive as responses from **kdp.ipr.praha.eu/api/foto/154** and can be placed into HTML img source tag directly. Backend side of things is as straightforward as one would expect. Function **retrieveFotoById(params)** was described above for sending HTTP headers and streaming picture's bytestream.

**index.js** of Node.js express backend, with app being instance of Express [8] running on sbds.domain.tld/**api**

```
app.get('/foto/:fotoId', async(req, res) => {
  res.send( await retrieveFotoById(req, res, req.params.fotoId))
})
```

**DisplayFoto.jsx** function component on React.js [9] frontend

```
import URLEndPointBE from './URLEndPointBE'
function DisplayFoto(props) {
  const fotoEndpoint = URLEndPointBE + `/foto`
  return(
    <span><img src={fotoEndpoint+"/"+props.foto.fotoId}</span>
  )
}
export default DisplayFoto
```

[8]https://expressjs.com

[9]https://reactjs.org

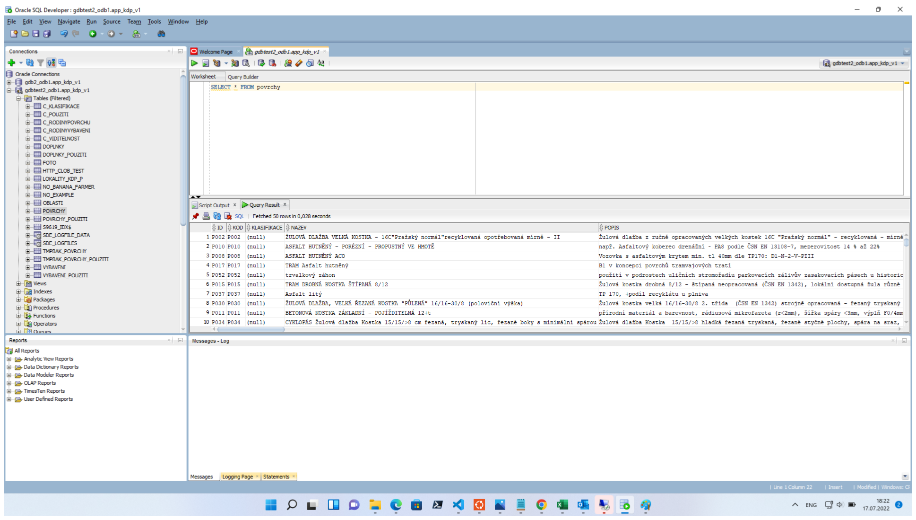
# 12 Oracle SQL Developer description

Going back to Oracle SQL Developer, there are some noteworthy things that have to be said about features and functions. These features facilitate process of working on database or viewing the data or even fiddling with it.

## Worksheet section

After opening Oracle SQL Developer and connecting to one database, a **Welcome page** pops up. It is notebook-worksheety tool, to execute SQL queries that are not necessarily affiliated to any specific part. One can, however, run queries or SQL statements to create functions or preform whatever actions that do (not) modify database structure/content.

Looking up, you can either **Run Statement (Ctrl+Enter)** or **Run Script (F5)** which differ in the readability of output, run statement returns aw data but run script returns you more verbous human-readable output.



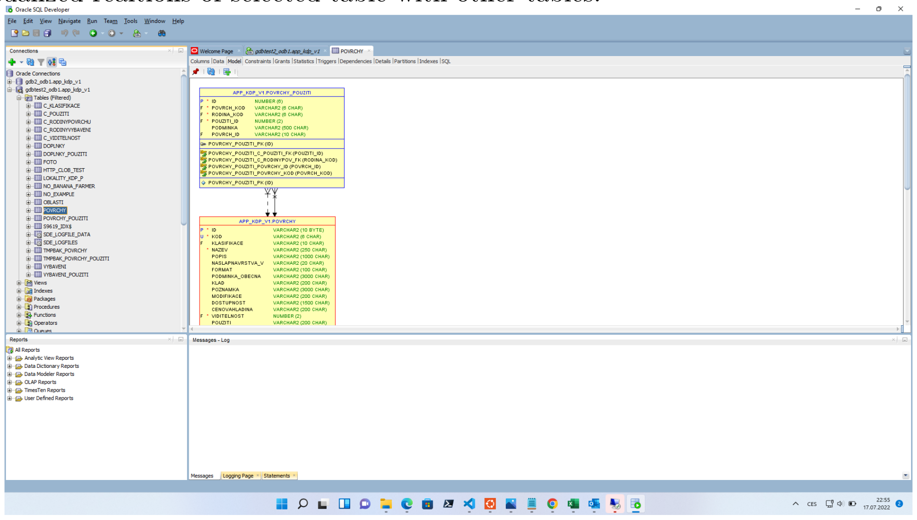
```
SELECT * FROM povrchy
```

Fetched 50 rows in 0,028 seconds

| | ID | KOD | KLASIFIKACE | NAZEV | POPIS |
|---|---|---|---|---|---|
| 1 | P002 | P002 | (null) | ŽULOVÁ DLAŽBA VELKÁ KOSTKA - 16C"Pražský normál"recyklovaná opotřebovaná mírně - II | Žulová dlažba z ručně opracovaných velkých kostek 16C "Pražský normál" - recyklovaná - mírně |
| 2 | P010 | P010 | (null) | ASFALT HUTNĚNÝ - PORÉZNÍ - PROPUSTNÝ VE HMOTĚ | např. Asfaltový koberec drenážní - PA8 podle ČSN EN 13108-7, mezerovitost 14 % až 22% |
| 3 | P008 | P008 | (null) | ASFALT HUTNĚNÝ ACO | Vozovka s asfaltovým krytem min. tl 40mm dle TP170: D1-N-2-V-PIII |
| 4 | P017 | P017 | (null) | TRAM Asfalt hutněný | B1 v koncepci povrchů tramvajových tratí |
| 5 | P052 | P052 | (null) | trvalkový záhon | použití v podrostech uličních stromořadiu parkovacích zálivův zasakovacích pásech u historic |
| 6 | P015 | P015 | (null) | TRAM DROBNÁ KOSTKA ŠTÍPANÁ 8/12 | Žulová kostka drobná 8/12 - štípaná neopracovaná (ČSN EN 1342), lokální dostupná žula různé |
| 7 | P037 | P037 | (null) | Asfalt litý | TP 170, +podíl recyklátu u plniva |
| 8 | P030 | P030 | (null) | ŽULOVÁ DLAŽBA, VELKÁ ŘEZANÁ KOSTKA "PŮLENÁ" 16/16-30/8 (poloviční výška) | Žulová kostka velká 16/16-30/8 2. třída (ČSN EN 1342) strojně opracovaná - řezaný tryskaný |
| 9 | P011 | P011 | (null) | BETONOVÁ KOSTKA ZÁKLADNÍ - POJÍŽDITELNÁ 12+t | přírodní materiál a barevnost, rádiusová mikrofazeta (r<2mm), šířka spáry <3mm, výplň F0/4mm |
| 10 | P034 | P034 | (null) | CYKLOPÁS Žulová dlažba Kostka 15/15/>8 cm řezaná, tryskaný líc, řezané boky s minimální spárou | Žulová dlažba Kostka 15/15/>8 hladká řezaná tryskaná, řezané styčné plochy, spára na sraz, |

## Table Model

Visualized realtions of selected table with other tables.

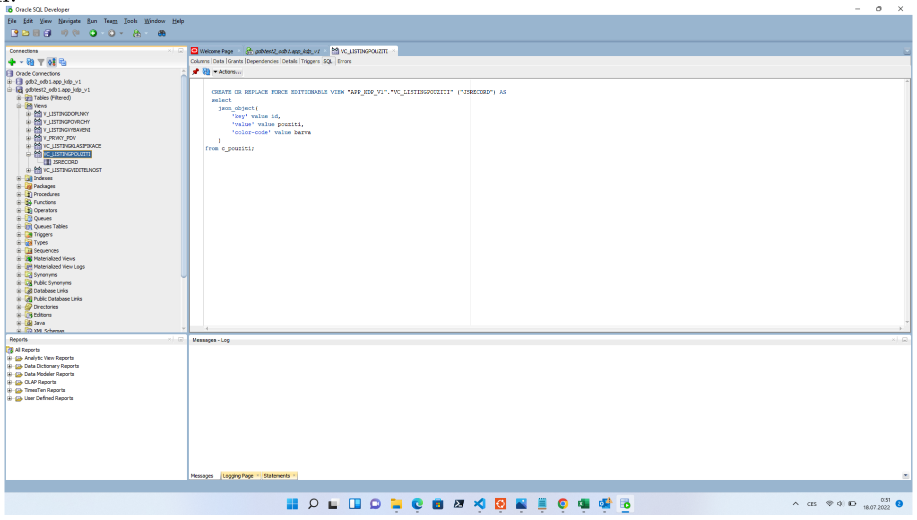


```
APP_KDP_V1.POVRCHY_POUZITI
P * ID              NUMBER (6)
F * POVRCH_KOD      VARCHAR2 (6 CHAR)
F * RODINA_KOD      VARCHAR2 (6 CHAR)
F * POUZITI_ID      NUMBER (2)
    PODMINKA        VARCHAR2 (500 CHAR)
F   POVRCH_ID       VARCHAR2 (10 CHAR)

POVRCHY_POUZITI_PK (ID)

POVRCHY_POUZITI_C_POUZITI_FK (POUZITI_ID)
POVRCHY_POUZITI_C_RODINYPOV_FK (RODINA_KOD)
POVRCHY_POUZITI_POVRCHY_ID (POVRCH_ID)
POVRCHY_POUZITI_POVRCHY_KOD (POVRCH_KOD)

POVRCHY_POUZITI_PK (ID)
```

```
APP_KDP_V1.POVRCHY
P * ID                  VARCHAR2 (10 BYTE)
U * KOD                 VARCHAR2 (6 CHAR)
F   KLASIFIKACE         VARCHAR2 (10 CHAR)
  * NAZEV               VARCHAR2 (250 CHAR)
    POPIS               VARCHAR2 (1000 CHAR)
    NASLAPNAVRSTVA_V    VARCHAR2 (20 CHAR)
    FORMAT              VARCHAR2 (100 CHAR)
    PODMINKA_OBECNA     VARCHAR2 (3000 CHAR)
    KLAD                VARCHAR2 (200 CHAR)
    POZNAMKA            VARCHAR2 (3000 CHAR)
    MODIFIKACE          VARCHAR2 (200 CHAR)
    DOSTUPNOST          VARCHAR2 (1500 CHAR)
    CENOVAHLADINA       VARCHAR2 (200 CHAR)
F * VIDITELNOST         NUMBER (2)
    POUZITI             VARCHAR2 (200 CHAR)
```

## Views SQL code example

Views editation preview in Oracle SQL Developer - img here, selection query as well.



```
CREATE OR REPLACE FORCE EDITIONABLE VIEW "APP_KDP_V1"."VC_LISTINGPOUZITI" ("JSRECORD") AS
select
  json_object(
      'key' value id,
      'value' value pouziti,
      'color-code' value barva
  )
from c_pouziti;
```