sciendo

# Containerization of Server Services

**doi:10.2478/mape-2020-0028**
Date of submission to the Editor: 03/2020
Date of acceptance by the Editor: 06/2020
**MAPE 2020, volume 3, issue 1, pp. 320-330**

**Adam Muc**
ORCID ID: 0000-0002-9495-087X

**Tomasz Muchowski**
ORCID ID: 0000-0002-0200-7041
Gdynia Maritime University, **Poland**

**Albert Zawadzki**
ORCID ID: 0000-0002-7516-0100
Polish Naval Academy of the Heroes of Westerplatte, **Poland**

**Adam Szeleziński**
ORCID ID: 0000-0003-2842-0683
Gdynia Maritime University, **Poland**

## INTRODUCTION

The growing popularity of remote working requires changes in the network infrastructure of enterprises. The changes also include the way web applications, databases and other network services are hosted. In local networks, in case of port conflicts, web applications and network services are hosted on separate servers. Separate servers also often host applications that rely on common dependencies but require different configuration. If enabling remote access to these services is needed, it may be necessary to perform port forwarding or equip servers with global IP addresses. If network traffic is allowed from an external network, the target device must be adequately secured, e.g. with firewall and antivirus software. This must be done for every device that can be accessed from an external network. This significantly increases the workload of the network administrator, as it is his task to maintain the server operating systems by updating and maintaining the enabled security measures. As the number of devices available from the external network increases, so does the probability of security failures and allowing data leakage or violation of the device's operating system by an unauthorized entity. This problem cannot be solved by purchasing a VPS server, as port collisions also occur in this case.

The solution to the problem may be the use of virtualization. A virtual machine has its own operating system and, with an appropriate network configuration it also has its own IP address in the local network (Portnoy, 2016). Network traffic can be redirected to virtual machines using port forwarding. Alternatively, the network cloud, i.e. Amazon AWS, can be used to create virtual machines with

Contact address: a.szelezinski@wm.umg.edu.pl

global IP addresses, in that case port forwarding is not necessary. However, the costs of network clouds can be high because the operating systems of each virtual machine create a resource overhead on the cloud. This situation also applies to the local server, whose resources will be used by the operating systems of the virtual machines it maintains. In addition, each of the virtual machines must be adequately secured so that they are not infected with malware or taken over by an unauthorized entity (Palmer, 2017; SUSE LLC, 2018).

A better solution may be to containerize the application. Containers create an isolated environment, but operate on the operating system of the parent device. Containerization software also allows to create virtual network adapters, which are used to isolate network connections made by containerized applications from the parent operating system and parent local network. This means that containerized applications can use the default network ports without interfering with other containerized applications if they operate on another virtual network infrastructure. They also do not interfere with the services of the host operating system. Virtual networks allow for port forwarding of the virtual network adapter so that the host operating system can access and share the container-maintained services on the local and external networks (Dua et. all, 2016; Poulton, 2017).

Containerization of services and applications completely solves port collisions and collisions of required configuration of common dependencies. An example of a containerization program is Docker. It is a program created in 2011 using Go programming language. Docker has gained a well-deserved popularity in the last two years and started to be used in many companies dealing with programming, DevOps and professional maintenance of network services (Container Journal Website, 2020). However, this software is not difficult to configure and can also be successfully used in small enterprises without IT support.

## TEST BECH

### Test server

The Docker engine can work on both Windows and Linux distributions because it allows containerization of software for both systems. It is not possible to run Linux containers natively on Windows and vice versa. The containers are linked to the operating system and cannot be run on a system incompatible with them (Nickoloff and Kuenzli, 2019; Poulton, 2017). By default, containers are built for use in Linux distributions, because containers built this way are more efficient than their Windows counterparts. The software native to Windows systems can also be containerized, but must be run on a device with Windows operating system installed (Stoneman, 2019).

Although native intersystem container use is not possible, Docker for Windows allows creating a lightweight Hyper-V virtual machine that will support Linux containers and provide access from the parent system to the opened ports of

the virtual network infrastructure, enabling communication with containerized applications. However, this is not an efficient solution because it requires emulation of the Linux kernel in the virtual machine. With the introduction of WSL2 – Windows Subsystem for Linux 2 (Microsoft Website, 2020) – it is possible to directly run Linux distributions and share resources with the native system and control them from Windows. This allows Linux containers to run on Windows in a WSL2 environment and get performance close to native. This is a good solution for development and testing environments, but in a production environment, for stability and native performance reasons, Linux containers should run on a server with a Linux distribution installed, e.g. Ubuntu Server, Debian or CentOS (Carter, 2017; Leszko, 2019).
In this case, the authors used WSL2 to test the containerized applications and check the operation of the created containers. Docker for Windows software can be downloaded from the manufacturer's website (Docker Documentation, 2020). After installation it is necessary to configure WSL2 – version 2004 of Windows 10 is required (Microsoft Website, 2020). Created containers can be run natively on any Linux distribution with support for the docker engine or in case of having another version of Windows it may be necessary to use Hyper-V. The Figure 1 below shows information about the version of software used by the authors.

```
C:\Users\DM>docker version
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:23:10 2020
 OS/Arch:           windows/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.8
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.17
  Git commit:       afacb8b
  Built:            Wed Mar 11 01:29:16 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          v1.2.13
  GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
```

**Fig. 1 View of Docker version**

## Test client devices

As client devices, the authors used two computers equipped with the Windows 10 operating system. One of them was both an application server and a client. In this case the connections were made using a localhost (loopback) address.

The second device was located in an external network to simulate requests coming from the Internet.

## WEB APPLICATION CONTAINERIZATION

### Standalone application

For testing purposes, the authors created a simple web application in Java using the Spring framework. The Figure 2 below shows a view of the project directory structure.

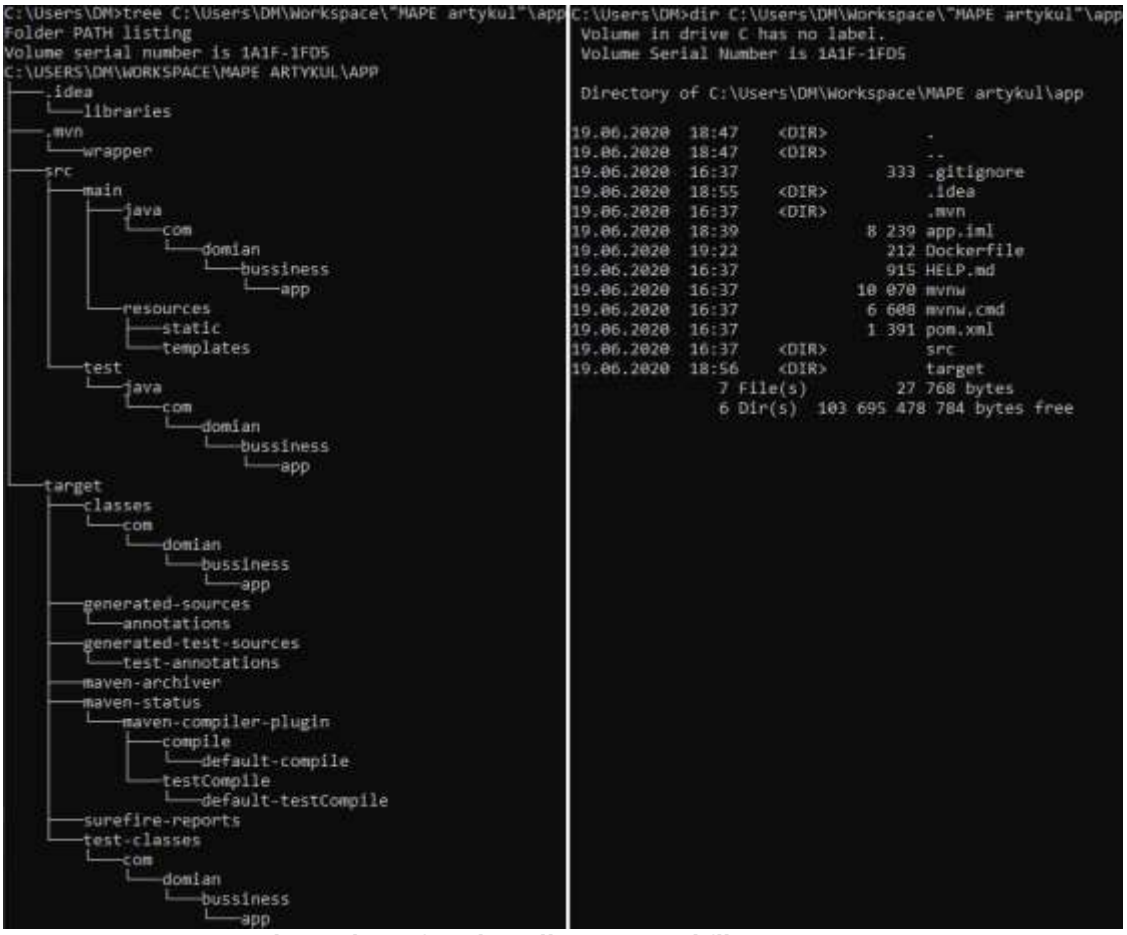


**Fig. 2 View of project directory and file structure**

In the Figure 3 above a Dockerfile file can be seen – it is a file that contains instructions for containerization process of the created application. The content of the Dockerfile is shown in the Figure below.

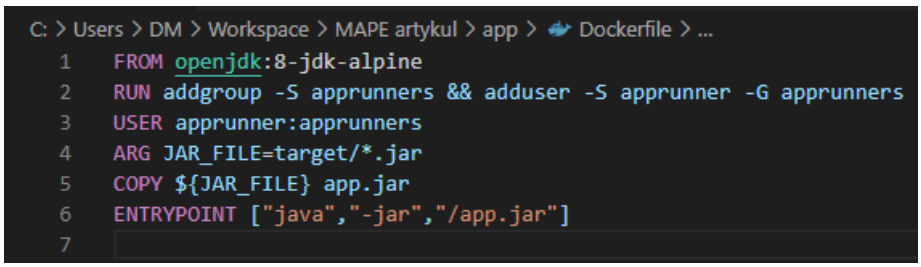


**Fig. 3 View of the Dockerfile**

As it can be seen in the figure above, this file contains a description of container image properties and a series of instructions. This image will be built from the *openjdk:8-jdk-alpine* image, because the JRE (Java Runtime Environment) is required to run an application created in Java. The selected OpenJDK image is based on the Alpine Linux image. Alpine Linux is a minimalist Linux distribution, occupying approximately 8MB of space (Alpine Linux Distribution Website, 2020), making it ideal for containerization use. The OpenJDK base image is open source and publicly available in the docker hub registry. It contains a ready to use and configured environment for Java applications. The image is then configured using a series of instructions written in the Dockerfile. A new user group is added and a new user is placed in it. This user will be used to run the application in the container. This is an optional step, but it is worth taking care that the application does not run with administrative privileges, as this may be a potential security breach. Then an argument is added. The arguments are variables that can be used in further lines of configuration code. In this case, the created argument is the path to the *.jar file, which should be placed in the container. In the next instruction, the file specified by the argument is copied to the container and saved under the app.jar name. Then the entry point is defined, which specifies what action the container should perform at the start. In this case it is going to launch the app.jar file. The created Dockerfile is ready for optimal test application containerization.

Before containerization, the application must be built, in case of Java applications, the resulting file is a file with *.jar extension. The Maven tool was used to build the test application. The next step is to issue a command to create container image. This operation is shown in the Figure 4 below.

```
C:\Users\DM\Workspace\MAPE artykul\app>docker build -t mucha17/bussiness-app-test .
Sending build context to Docker daemon  16.75MB
Step 1/6 : FROM openjdk:8-jdk-alpine
 ---> a3562aa0b991
Step 2/6 : RUN addgroup -S apprunners && adduser -S apprunner -G apprunners
 ---> Using cache
 ---> 8e54e63e5554
Step 3/6 : USER apprunner:apprunners
 ---> Using cache
 ---> ab7cf56a7c62
Step 4/6 : ARG JAR_FILE=target/*.jar
 ---> Using cache
 ---> b67e9026e107
Step 5/6 : COPY ${JAR_FILE} app.jar
 ---> 45c23fe43ba2
Step 6/6 : ENTRYPOINT ["java","-jar","/app.jar"]
 ---> Running in 61fa37638158
Removing intermediate container 61fa37638158
 ---> cd4d14f86811
Successfully built cd4d14f86811
Successfully tagged mucha17/bussiness-app-test:latest
```

**Fig. 4 Process of building container image**

A container image will be created, which can be used to launch a new container with the test application. The image list can be displayed using the docker image list command. The newly created image is only available on the computer on which it was created. The image can be uploaded the docker hub registry to be used on other devices as well. However, it is worth mentioning that the docker hub is publicly accessible to everyone. If the application is intended for specific company use, it may be necessary to create a private registry. In this case, one of the authors' account in the docker hub registry was used. The publication of the image is an optional activity. The created image can be used locally to create containers. Publication is necessary if containers with the application will be created on other devices as well, because they must have access to the image in order to create a container. The following Figure 5 shows the process of displaying available images, publishing an image and creating a container based on the created image.


**Fig. 5 Creating a container**

As it can be seen in the figure above, a container was created and given the name BussinessAppTest. The application uses port 8080 by default, but the container's traffic on this port has been redirected to port 80 of the parent system. This means that the parent system can display the application's network content because the traffic was redirected to its interface on port 80. As the containerized application is a web application, a web browser should be used to display its network content. The following Figure 6 shows an attempt to display the web content of a containerized web application.

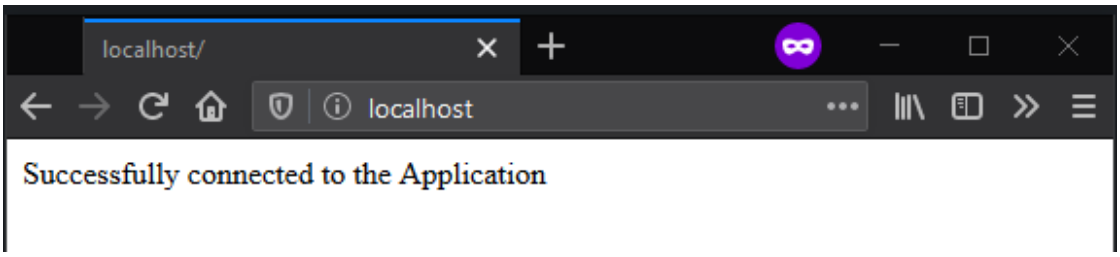


**Fig. 6 Test of access to the web application from the server's web browser**

As it can be seen in the Figure above, the network content of the containerized application is correctly forwarded to the network interface of the parent operating

system. This means that if the server has a public IP address and the firewall is properly configured, the application is also available on the external network. In this case, the server does not have a public IP address, but port forwarding was used to redirect incoming network traffic from the external network to a given port of the server's network interface. A second test device was placed in the external network and was used to connect to the router's DNS address. The operation is shown in the Figure 7 below.

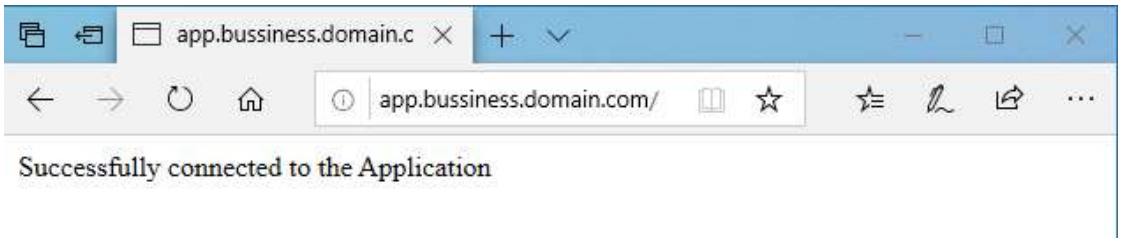


**Fig. 7 Test of access to the web application from the external network**

As it can be seen in the figure above, the operation was successful. It means that the application has been properly containerized and properly isolated from the parent system.

**Application using dependencies**

The application from the previous example did not communicate with other containers and did not have any dependencies like data from the database. The default virtual network adapter was used, bridging the virtual interface with the network interface of the parent system.

Applications increasingly require communication with other applications on which they depend. An example of dependency can be a database. Databases are maintained by means of applications called DBMS (Data Base Management System). DBMS can maintain many databases and theoretically can serve data to many applications simultaneously. However, very often it happens that applications require DBMS to have a custom configuration or require a specific version of DBMS. In this case, there is a conflict that requires the network administrator to reconfigure each application dependent on DBMS so that their configuration is compatible with each other. However, this is not always possible, because some applications are not equipped with the option to change network configuration. In this case the configuration conflict cannot be solved and it is necessary to install the second DBMS instance in a different location (on a different server or in a virtual machine). Maintaining multiple servers or virtual machines, just because the application configuration requires it, is not optimal. This problem is solved by containerization, because the docker isolates all containers from each other by default. Creating virtual network adapters allows the exchange of network data between containers connected to the same network adapter. This means that the administrator can completely isolate applications from each other and create separate virtual network adapters for

them. Then connect a separate dependency instance for each application to each virtual network adapter.

The authors have containerized a web application using the MongoDB database as a dependency. The authors decided to create a virtual network adapter to which a container with a web application and a container with MongoDB DBMS was connected. A virtual volume was also created, in which all MongoDB data will be stored. In case of a DBMS container failure, data from the database will not be lost and can be read by another container created as a replacement for the damaged container. The Figure 8 below shows the process of creating a virtual network adapter, virtual volume for the MongoDB database and launching containers using the created virtual adapter.

```
C:\Users\DM>docker network create bussiness-app-network
01b6e16681dc057397e4af33d16c524ab2759fd8df9ba3c422deb8fc4ff93cb7

C:\Users\DM>docker volume create bussiness-app-data
bussiness-app-data

C:\Users\DM>docker run -d --name bussiness-app-mongodb --net bussiness-app-net
work -v bussiness-app-data:/data/db mongo:latest
6e7f3c1c58bdd48e3583680affa29001c393c6c51ff0fdd336855814fa69d809

C:\Users\DM>docker run -d --name bussiness-dbbackend-test --net bussiness-app-
network -p 8080:8080 mucha17/bussiness-dbbackend-test
582f21adfd109e3300b5c889be9c26b2461856493c5d3ad0bc067a117fb7f1b8
```

**Fig. 8 Creating a virtual network adapter, volume and containers**

The creation process must always be verified. The following Figure 9 shows the process of verifying whether an virtual adapter, volume and containers have been created.

```
C:\Users\DM>docker container list
CONTAINER ID   IMAGE                                COMMAND                  PORTS                    NAMES
582f21adfd10   mucha17/bussiness-dbbackend-test     "java -jar /app.jar"     0.0.0.0:8080->8080/tcp   bussiness-dbbackend-test
6e7f3c1c58bd   mongo:latest                         "docker-entrypoint.s…"   27017/tcp                bussiness-app-mongodb

C:\Users\DM>docker network list
NETWORK ID     NAME                   DRIVER    SCOPE
0ea5f34d69ea   bridge                 bridge    local
01b6e16681dc   bussiness-app-network  bridge    local
2888ec3e31c7   host                   host      local
117602f42841   none                   null      local

C:\Users\DM>docker volume list
DRIVER    VOLUME NAME
local     2df54a38ef2f5191506c1337441010bf99db37b7e59556c3702578659d96cf4c
local     32974db0ef90d9b4cf4d23159437b78223faf9c12437eab5804fe0f4b59a634d
local     b83f7d0150e3b9950eaa8ef4f7390d09f4e095596c864088cc3d8a52a26c3d29
local     bussiness-app-data
local     df03ffba9870e4aa0ffa98c55e7087d1770d643c9180e214930d0bddfff5a15f1
```

**Fig. 9 View of list of created virtual adapters, volumes and containers**

As it can be seen in the figure above, the operation of creating a virtual network adapter, virtual volume and containers was successful. It should also be noted that the virtual adapter ports for the DBMS container were not forwarded. This means that communication with this container is only possible from within the virtual network infrastructure. This provides complete isolation and guarantees that only the containerized application can access its dependency.

As the virtual adapter port for the container with the web application has been forwarded, it is available from the parent operating system. It should be noted

that the application occupies the 8080 port of the parent system and the application from the first example occupies the 80 port. By default, both applications work on port 8080 and there would be a collision, but thanks to containerization and appropriate port forwarding, both applications can work simultaneously. The following Figure 10 shows a connection test to the containerized web application.

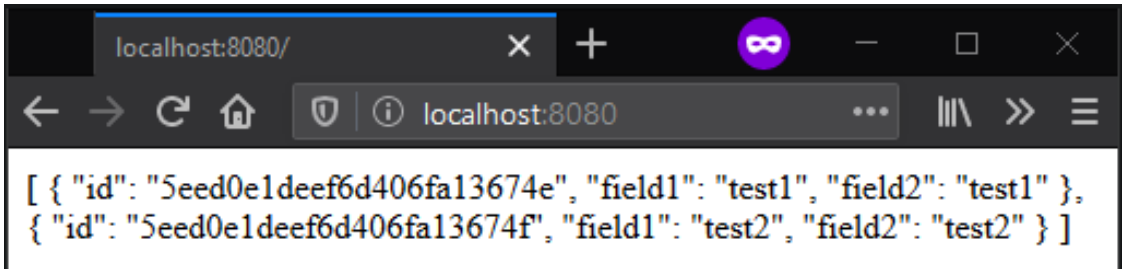


**Fig. 10 Connection test to the containerized application**

As it can be seen in the figure above, the application returns the collection contents from the database. The test application is an application that removes the collection when the page is loaded, and then creates it again by creating new documents. Then the data is send and displayed to the user. This is a typical application testing the correctness of connection to the database and the functionality of CRUD operations (Create, Read, Update, Delete). The following Figure 11 shows the source code fragment responsible for testing the database.

```
@RequestMapping("/")
public String home() {
    this.mongoTemplate.dropCollection(TestDocument.class);
    TestDocument testDocument1 = new TestDocument("test1", "test1");
    TestDocument testDocument2 = new TestDocument("test2", "test2");
    List<TestDocument> list = Arrays.asList(testDocument1, testDocument2);
    this.testRepository.saveAll(list);
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    return gson.toJson(this.testRepository.findAll());
}
```

**Fig. 11 View of virtual tunneling interface configuration**

The same Dockerfile as the previous application was used to create an image of the web application. By using arguments, it is universal for applications created in Java programming language. By creating an appropriate Dockerfile it is possible to containerize most application and its dependencies.

## CONCLUSION

Containerization consists in isolating the software from applications installed in the parent operating system. Containerized software does not cause collision of used network ports and dependency configuration like DBMS configuration. Appropriate creation of Dockerfile configuration file allows to create container

image of most applications. Using the image containers can be created. Images can be used locally or published to a registry. The Docker Hub is the largest container registry, but as it is a public registry, use of private registry may be a better fit for company applications container images.

Software isolation solves the problem of collisions in a simple and cost effective way. It is better than alternatives such as using virtual machines because virtual machines create an overhead of used resources by the necessity of having an operating system installed. This also means extra work with maintenance. Containerization solves this problem. Containers are also safer to use, as they do not require separate security features. A well-secured server's host operating system will prevent the integrity of the containers from being compromised or data stolen from the containerized application. Due to numerous advantages in comparison with virtual machines, the containerization technology can be used in companies that are equipped with servers that host network services.

## REFERENCES

Carter, T. (2017) Docker and virtual machines. Independently published

Dua, R. and Kohli, V. and Konduri, S.K. (2016) Learning Docker Networking. Birmingham: Packt Publishing.

Alpine Linux Distribution Website. About section [online]. Available at: https://alpinelinux.org/about/ [Accessed 1 June 2020].

Container Journal Website. Using Google trends to chart Docker's rise to fame [online]. Available at: https://containerjournal.com/features/using-google-trends-chart-dockers-rise-fame/ [Accessed 1 June 2020].

Docker Documentation. Docker Desktop for Windows [online]. Available at: https://docs.docker.com/docker-for-windows/install/ [Accessed 1 June 2020].

Microsoft Website. Windows Documentation – WSL2 [online]. Available at: https://docs.microsoft.com/en-us/windows/wsl/wsl2-index [Accessed 1 June 2020]

Leszko, R. (2019) Continuous Delivery with Docker and Jenkins, 2nd Edition. Birmingham: Packt Publishing.

Nickoloff, J. and Kuenzli, S. (2019) Docker in Action, 2nd Edition. Manning Publications.

Palmer, M. (2017) Hands-On Microsoft Windows Server 2016, 2nd Edition. Boston: Cengage Learning.

Portnoy, M. (2016) Virtualization Essentials, 2nd Edition. New York: Sybex.

Poulton, N. (2017) Docker Deep Dive. Independently published.

Stoneman, E. (2019) Docker on Windows, 2nd Edition. Birmingham: Packt Publishing.

SUSE LLC (2018) SUSE Linux Enterprise Server 12 - Virtualization Guide. Suwanee: 12th Media Services.

**Abstract:** Businesses are increasingly confronted with server-related problems. More and more, businesses are enabling remote working and need to rely on network services. The provision of network services requires rebuilding the network infrastructure and the way employees are provided with data. Web applications and server services use common dependencies and require a specific network configuration. This often involves collisions between network ports and common dependencies' configuration. This problem can be solved by separating the conflicting applications into different servers, but this involves the cost of maintaining several servers. Another solution may be to isolate applications with virtual machines, but this involves a significant overhead on server resources, as each virtual machine must be equipped with an operating system. An alternative to virtual machines can be application containerization, which is growing in popularity. Containerization also allows to isolate applications, but operates on the server's native operating system. This means eliminating the overhead on server resources present in virtual machines. This article presents an example of web application containerization.

**Keywords:** remote work, server services, containerization, Docker, server costs