Assignment 5

by

Adam Hazel

in

IKT206

DevOps

Grimstad, 18/03/25

List of Figures

List of Listings

Introduction

The following lab consists of three tasks which should help the student understand how to host an ASP.NET application on a cloud Virtual Private Server with the HTTPS protocol.

In completion of the lab exercise, the student should be able to:

1. Create an ASP.NET project that will be hosted on the VPS.
2. Create the needed Docker Compose files for building and hosting the application.
3. Deploy the VPS.

This report explains the steps taken to complete the given tasks so that others would be able to reproduce the same results.

1 ASP.NET Application - Code preparation

1.1 Cloning repository and copying codebase

The first step of the task is to create a new repository on our Gitea server (which is hosted on the VM provided to us). After creating the repository on our Gitea server, we clone it to our VM to begin working on the project..

As the ssh protocol uses port 22, but, in our VM, we have set up port forwarding so that port 22 is accessed through port 222, we need to edit the `git clone` command to include the port number as follows:

```
1 git clone ssh://git@10.225.150.141:222/arhazel/ikt206g25v-05.git
```

Listing 1: Git clone command with specified port number

To add code to the cloned repository, we can access the remote VM through VSCode using the "Remote-SSH" extension as seen in figure 1.

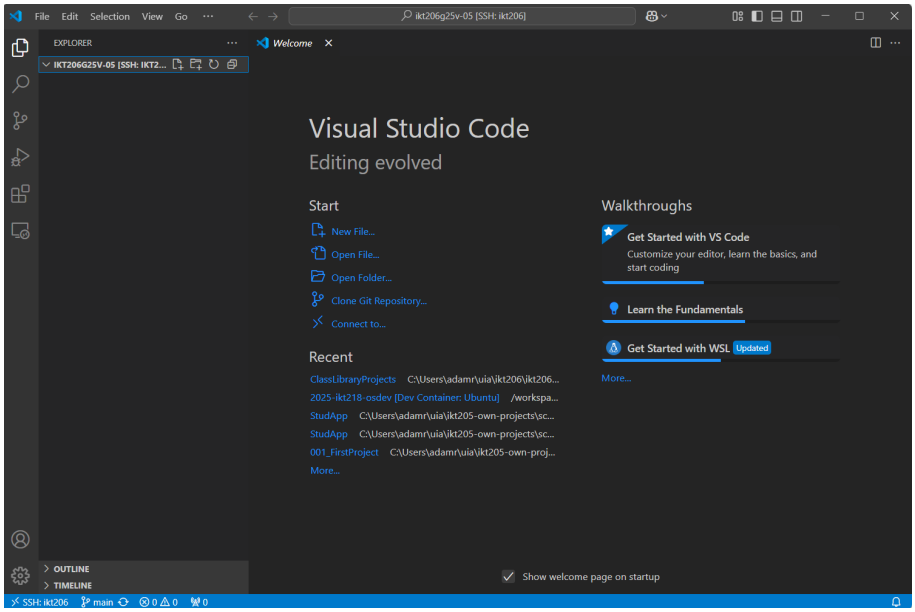

Figure 1: Using VSCode to access the repository on the VM

1.2 Installing dotnet on VM

To install dotnet on the Virtual machine, we run the following commands:

```
wget https://dot.net/v1/dotnet-install.sh
chmod +x dotnet-install.sh
./dotnet-install.sh --channel 8.0
```

Listing 2: Command to get the dotnet install script and execute it

```
echo 'export DOTNET_ROOT=$HOME/.dotnet' >> ~/.bashrc
echo 'export PATH=$DOTNET_ROOT:$DOTNET_ROOT/tools:$PATH' >> ~/.bashrc
source ~/.bashrc
```

Listing 3: Adding dotnet to PATH so that it can be executed in the terminal

We then add the needed packages to the dotnet solution:

```
dotnet add package Microsoft.EntityFrameworkCore --version 8.0.2
dotnet add package Microsoft.EntityFrameworkCore.Design --version 8.0.2
dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 8.0.2
dotnet add package Microsoft.EntityFrameworkCore.Tools --version 8.0.2
dotnet add package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore --version 8.0.2
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 8.0.2
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL --version 8.0.2
```

Listing 4: Add needed packages to the dotnet solution

```
dotnet restore
```

Listing 5: Running dotnet restore to download and install dependencies

```
dotnet tool install --global dotnet-ef
export PATH="$HOME/.dotnet/tools:$PATH"
echo 'export PATH="$HOME/.dotnet/tools:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Listing 6: Add the Entity Framework Command Line interface to PATH so that adding migrations via. terminal is possible

Without the code in listing 3 and listing 6, we received errors when trying to invoke dotnet commands. We also remove the dotnet-install.sh script after installation.

1.3 Editing code for production mode

appsettings.json

In this file, we added the another item to the "ConnectionStrings" object so that there is a key-pair for "DefaultConnection" and "ProductConnection". The "ProductConnection" contains the configuration for the Postgres database.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "DataSource=app.db;Cache=Shared",
    "ProductionConnection":
    ↪  "Host=postgres;Port=5432;Username=postgres;Password=Password1;Database=postgres;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Listing 7: appsettings.json with added connection string

1.3.1 Program.cs

Initially in Program.cs, a default database service is registered as seen in listing 8. This is expanded to register the needed service when the project runs in production mode as seen in listing 9.

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlite(connectionString));
```

Listing 8: Initial database service registration

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Npgsql.EntityFrameworkCore.PostgreSQL;

using Example.Data;

var builder = WebApplication.CreateBuilder(args);

// Register ApplicationDbContext with environment-specific configuration
if (builder.Environment.IsDevelopment())
{
    builder.Services.AddDbContext<ApplicationDbContext>(options =>
        ↪ options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));
}
else
{
    builder.Services.AddDbContext<ApplicationDbContext>(options =>
        ↪ options.UseNpgsql(builder.Configuration.GetConnectionString("ProductionConnection")));
};
```

Listing 9: Configured database service registration

2 Docker and Docker Compose

2.1 Preparing project for containerization

The aim is to build our dotnet project to an image so that we can run containers based on that image. In order to create an image, we need to use a Dockerfile.

2.1.1 Dockerfile

According to the Microsoft documentation, the Dockerfile should be in the same working directory as the *.csprosj* file.

The Dockerfile contains code as seen in listing 10. The process mapped out by the Dockerfile uses multi-stage builds [1], where each stage starts with a `FROM` command and only copied artifacts are taken over into the next stage. The steps are as follows:

1. **First stage:**
   (a) Get a dotnet SDK 8.0 base image that will be used to compile the project.
   (b) Create a working directory called "app".
   (c) Copy the contents of current local directory (where the Dockerfile is contained" to the newly created /app directory.
   (d) Restore project dependencies.
   (e) Publish the project (bundling all created files into the out directory.
2. **Second stage:**
   (a) Get a specific dotnet image for running the project
   (b) Create a new working directory and copy the published "out" directory from the first stage.
   (c) Define the entrypoint as running the dotnet command `dotnet Example.dll`

```
# Use .NET SDK to build the project
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /app

# Copy the project to the container working directory
COPY . ./
# Restore the project dependencies
RUN dotnet restore
# Build the project in release mode
RUN dotnet publish -c Release -o out

# Build runtime image
FROM mcr.microsoft.com/dotnet/aspnet:8.0@[specific_checksum]
WORKDIR /app
COPY --from=build /app/out .
ENTRYPOINT ["dotnet", "Example.dll"]

```

Listing 10: Dockerfile where specific checksum is removed for conciseness

2.1.2 Testing Dockerfile with temporary container

We can test that the Dockerfile works by building the Dockerfile using the command given in listing 11. We can then see that an image has been created by using the command given in listing 12, the result being given in figure 2.

```
docker build -t myapp .
```

Listing 11: Command to build Dockerfile

```
docker images ls
```

Listing 12: Command to list Docker images

```
arhazel@ikt206-g-25v-arhazel:~$ docker image ls
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
myapp               latest    db1171528a22   32 minutes ago   312MB
alpine              latest    aded1e1a5b37   4 weeks ago      7.83MB
hello-world         latest    74cc54e27dc4   7 weeks ago      10.1kB
gitea/gitea         1.23.1    5774a23fffdb   2 months ago     177MB
gitea/act_runner    latest    4a7531b6e6b4   5 months ago     41.5MB
```

Figure 2: Result of using the command given in listing 12

We also decided to test that the container was running by using the command given in listing 13. The visual proof is seen in figure 3.

```
arhazel@ikt206-g-25v-arhazel:~/ikt206/ikt206g25v-05/Example$ docker run --rm -e
↪  ASPNETCORE_ENVIRONMENT=Development -p 9000:8080 myapp
```

Listing 13: Command to start temporary image in development mode

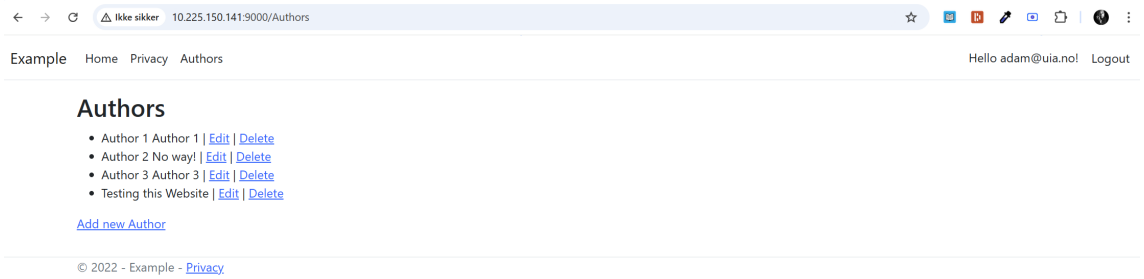

Figure 3: Running container in development mode

2.2 Creating a network

We wish to create a network that all the services will use. First, we create a directory that will contain all the services, and, in this directory, we create a docker-compose file specifically for networks as seen in listing 14 (the file has a dummy service just for testing). Proof that this works is shown in figure 4. The name of the network is given in line 3. We use the name on line 2 in other compose files to connect to this network.

```
1  networks:
2    dotnet_app_network:
3      name: dotnet_app_network
4
5  services:
6    dummy:
7      image: busybox
8      command: sleep infinity
9      networks:
10     - dotnet_app_network
```

Listing 14: Compose file for setting up a network

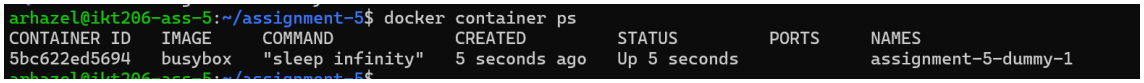

Figure 4: Proof of creating network and dummy service

2.3 Creating the database

Now that the network is created, we set up the Postgres database. Listing 15 provides the complete code for the compose file.

A simple explanation of each part of the code is now given:

- **services**: Details regarding which services to containerize
  - image: from which image the container will be built
  - container_name: how the container will be referenced to by Dcoker
  - restart: the restart policies when the container stops. Here, we state that the container app_db should be restarted unless stopped manually
  - environment: these are the environment variables used to access the database. These, including the container name, must be the same as described in Program.cs (see listing 7).
  - networks: the network that the container can use. This same network is described in the networks top-level statement, which states that this network already exists (created in listing 14)
  - ports: here, the port mapping is given. Postgres database uses port 5432.
  - volumes: this is where permanent data is stored on the host machine that is mapped to the data in the container. This volume is also created in the top-level statement "volumes". A name is explicitly stated.
- **networks**: This is the top-level statement that defines the network that the service connects to.
- **volumes**: This is the top-level statement that defines the volume that the service connects to.

```yaml
services:
  app_db:
    image: postgres:latest
    container_name: postgres_db
    restart: unless-stopped
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: Password1
      POSTGRES_DB: mydatabase
    networks:
      - dotnet_app_network
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

networks:
  dotnet_app_network:
    external: true

volumes:
  pgdata:
    name: pgdata
```

Listing 15: Compose file for setting up a PostgresSQL database

We can run the `docker network inspect [NETWORK_NAME]` to see which containers are connected to the network. The result is shown in figure 5. We can also see that the volume has been created as seen in figure 6.

Figure 5: Screenshot of containers connected to created network

```
arhazel@ikt206-ass-5:~/assignment-5$ docker volume ls
DRIVER    VOLUME NAME
local     pgdata
```

Figure 6: Confirmation that volumes exist

2.4 Adding our .NET app

2.4.1 Testing on development mode

For information on cloning project directory to VPS, see section 3.3.

Having cloned our project files, we first write a simple Docker Compose file as follows:

```yaml
services:
  app:
    container_name: app
    build:
      context: ./app/Example
      dockerfile: Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
    ports:
      - "3232:8080"
    networks:
      - dotnet_app_network
    depends_on:
      - app_db
    restart: unless-stopped

networks:
  dotnet_app_network:
    external: true

volumes:
  pgdata:
    name: pgdata
```

Listing 16: Compose file for setting up the app

IMPORTANT: We moved all the compose files to the root of the project directory as seen in listing 17.

```
arhazel@ikt206-ass-5:~/assignment-5$ ls -la
total 24
drwxrwxr-x 3 arhazel arhazel 4096 Mar 21 08:24 .
drwxr-x--- 9 arhazel arhazel 4096 Mar 20 20:45 ..
drwxrwxr-x 4 arhazel arhazel 4096 Mar 21 08:24 app
-rw-rw-r-- 1 arhazel arhazel  351 Mar 21 08:36 docker-compose-app.yml
-rw-rw-r-- 1 arhazel arhazel  429 Mar 19 08:28 docker-compose-db.yml
-rw-rw-r-- 1 arhazel arhazel  166 Mar 18 09:50 docker-compose-net.yml
```

Listing 17: Project structure on the VPS

Comments on listing 16:

- container_name: name assigned to container
- build: the container should be built from the Dockerfile found in the path given by "context"
- environment: Environment variables to build the container in production mode
- ports: An arbitrary port mapping to access the running app
- networks: The networks the container has access to
- depends_on: The running of "app" is dependent on the running of the Postgres container
- restart: the restart policy is to restart unless manually stopped

We run the containers using the command given in listing 18. The flag `--build` forces Docker to build the .NET project from scratch (which is useful if we have changed code within the project). Figure 7 shows the app running.

```
docker compose -f docker-compose-app.yml -f docker-compose-db.yml up --build -d
```

Listing 18: Docker compose up command for building app and db containers

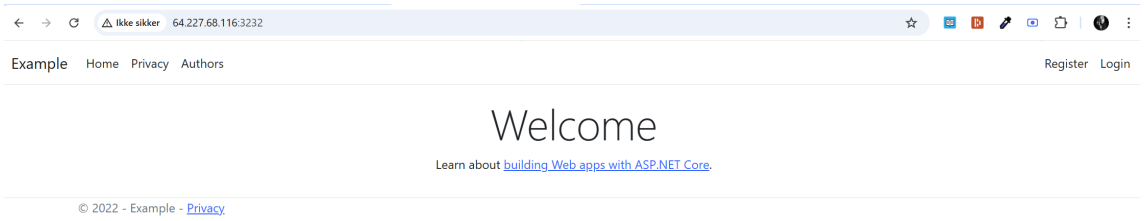

Figure 7: App running in production mode

3 Server deployment

3.1 Domain purchase

Proof of domain purhcase is given in figure 8.

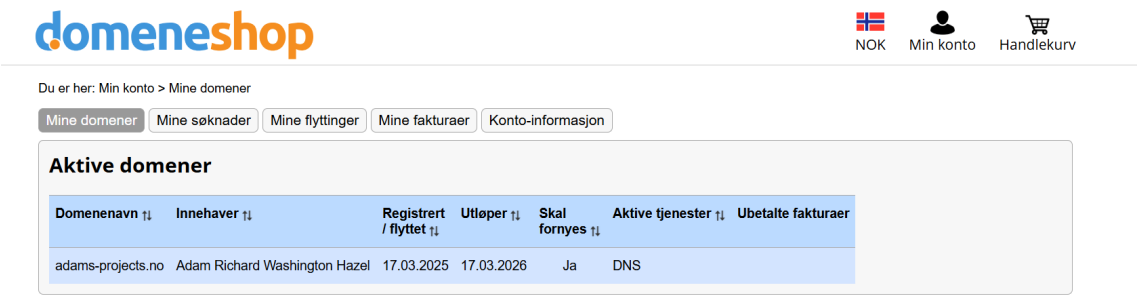

Figure 8: Proof of domain purchase

3.2 Setting up user on VPS

To separate security concerns, we can create a new user on the VPS. The steps are as follows:

1. Create a user (19)
2. Elevate privileges assigned to user (20)

```
1 sudo adduser arhazel
```

Listing 19: Creating new user

```
1 sudo usermod -aG sudo arhazel
```

Listing 20: Elevating privileges of user

3.3 Setting up reverse tunneling

As access to the school VM is blocked by the school's firewall, then we can use **reverse tunneling** to access files on the school VM and copy to the VPS. "Reverse SSH tunneling allows you to use that established connection to set up a new connection from your local computer back to the remote computer" [2].

The goal here is to connect to the VPS from the school machine, and then use that established connection to connect back from the VPS to the school machine.

First, we generated a ssh key pair on the school VM and copied the public key to the VPS (as seen in listing 21 thus allowing us to log on to the VPS from the school machine.

After ensuring that we can now log on to the VPS from the school machine, we run the command given in listing 22, which sets up a port mapping so that requests sent to port 43022 on the VPS are forwarded to port 22 (the port for ssh) on the school VM, and requests send to port 43222 on the VPS are sent to port 222 on the school VM (which is where Gitea is listening). It is important that the terminal stays open so that the tunnel remains open.

From the VPS, we run the command given in listing 23. This tells Git to connect to port 43222 on the VPS, using the ssh-key referenced to by the -i flag. The port mapping from reverse tunnel connects us to port 222 on the VM, which again is mapped to Gitea server port 22. The request is then processed by Gitea and the result is sent back through the tunnel, allowing the repository to be cloned on the VPS.

```
1 ssh-keygen -t ed25519 -C "ikt206-SchoolVM" -f ~/.ssh/id_ass_5_VPS
```

Listing 21: Creating a ssh-key pair for logging on to the VPS from the school VM

```
1 ssh -i ~/.ssh/id_ass_5_VPS -R 43022:localhost:22 -R 43222:localhost:222
  ↪ arhazel@64.227.68.116
```

Listing 22: Setting up a reverse tunnel between school VM and VPS

```
1 GIT_SSH_COMMAND="ssh -i ~/.ssh/id_gitea_ikt206 -p 43222" git clone
  ↪ ssh://git@localhost/arhazel/ikt206g25v-05.git .
```

Listing 23: Cloning using reverse tunneling whilst in the directory "app"

To be able to push changes back to Gitea, we need to change our git remote as seen in listing 24.

```
1 git remote set-url origin ssh://git@localhost:43222/arhazel/ikt206g25v-05.git
```

Listing 24: Updating git to a new remote

When pushing to the Gitea server, we still need to have the reverse tunnel open. We can then use the command as given in lisitng 25.

```
1 GIT_SSH_COMMAND="ssh -i ~/.ssh/id_gitea_ikt206 -p 43222" git push origin main
```

Listing 25: Updating git to a new remote

We can now run the commands shown in listing 11 and listing 13 under section 2.1.2. The result should be that we can access the running project using the VPS IP address and port number 8080.

3.4 DNS-Pointers

Figure 9 shows how the we use DNS pointing to access the app. Figure 10 shows the app running, but without the HTTPS protocol.

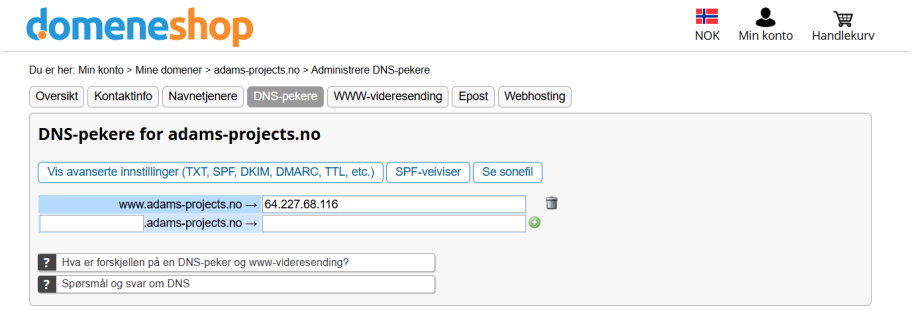

Figure 9: DNS pointing to VPS

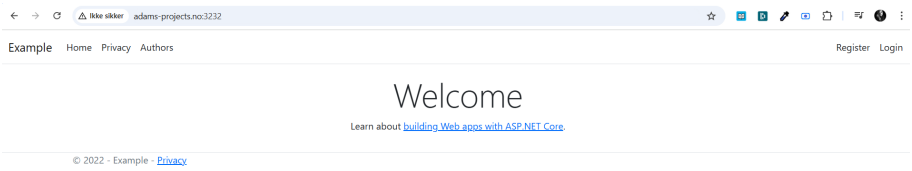

Figure 10: Website working without HTTPS protocol

3.5 Setting up Caddy

Caddy is a reverse proxy that automatically handles HTTPS certificates.

We begin with the compose file for the Caddy container which is provided in listing 26. The ports attribute maps the ports of the VPS to the ports of the Caddy container. Without this, we would not be accessing the Caddy container (and thus using the reverse proxy).

The Caddyfile details what services from Caddy we will use. This is given in listing 27. We use the reverse-proxy service to handle requests going to the domain, and forward them to the container called "app" at port 8080 (that is, the port inside the container).

```yaml
services:
  caddy:
    image: caddy:2
    container_name: caddy
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./caddy/data:/data
      - ./caddy/config:/config
      - ./caddy/Caddyfile:/etc/caddy/Caddyfile
    networks:
      - dotnet_app_network

networks:
  dotnet_app_network:
    external: true
```

Listing 26: Compose file for Caddy container

```
www.adams-projects.no {
    reverse_proxy app:8080
}
```

Listing 27: Contents of Caddyfile

We make final adjustments by removing the port mapping from 16 in section 2.4.1 so that the only way to access the app is through Caddy reverse proxy.

Figure 11 shows the app running through the domain, with a https certificate.

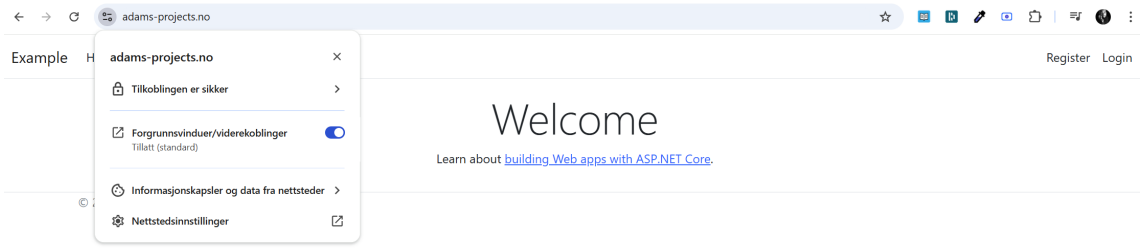

Figure 11: App running with HTTPS protocol

References

[1] *"Multi-stage"*, [Online; accessed 17. Mar. 2025], Sep. 2024. [Online]. Available: `https://docs.docker.com/build/building/multi-stage`.

[2] D. McKay, "What Is Reverse SSH Tunneling? (and How to Use It)," *How-To Geek*, Sep. 2023. [Online]. Available: `https://www.howtogeek.com/428413/what-is-reverse-ssh-tunneling-and-how-to-use-it`.