

TUTORIAL

How To Remotely Access GUI Applications Using Docker and Caddy on Ubuntu 18.04

2 June 2020

Introduction

Even with the growing popularity of cloud services, the need for running native applications still exists.

By using [noVNC](#) and [TigerVNC](#), you can run native applications inside a [Docker](#) container and access them remotely using a web browser. Additionally, you can run your application on a server with more system resources than you might have available locally, which can provide increased flexibility when running large applications.

In this tutorial, you'll containerize [Mozilla Thunderbird](#), an email client, using Docker. Afterward, you'll secure it and provide remote access using the [Caddy](#) web server.

When you're finished, you'll be able to access Thunderbird from any device using just a web browser. Optionally, you'll also be able to locally access the files from it using [WebDAV](#). You'll also have a fully self-contained Docker image that you can run anywhere.

Prerequisites

Before you begin this guide, you'll need the following:

- One Ubuntu 18.04 server with at least 2GB RAM and 4GB disk space.
- A non-root user with `sudo` privileges.

- Docker set up on your server. You can follow the [How To Install and Use Docker on Ubuntu 18.04](#).

Step 1 — Creating the `supervisord` Configuration

Now that your server is running and Docker is installed, you are ready to begin configuring your application's container. Since your container consists of multiple components, you need to use a process manager to launch and monitor them. Here, you'll be using `supervisord`. `supervisord` is a process manager written in Python that is often used to orchestrate complex containers.

First, create and enter a directory called `thunderbird` for your container:

```
mkdir ~/thunderbird
cd ~/thunderbird
```

Now create and open a file called `supervisord.conf` using `nano` or your preferred editor:

```
nano supervisord.conf
```

Now add this first block of code into `supervisord.conf`, which will define the global options for `supervisord`:

```
~/thunderbird/supervisord.conf
```

```
[supervisord]
nodaemon=true
pidfile=/tmp/supervisord.pid
logfile=/dev/fd/1
logfile_maxbytes=0
```

In this block, you are configuring `supervisord` itself. You need to set `nodaemon` to `true` because it will be running inside of a Docker container as the entrypoint. Therefore, you want it to remain running in the foreground. You also are setting `pidfile` to a path accessible by a non-root user (more on this later), and `logfile` to `stdout` so you can see the logs.

Next, add another small block of code to `supervisord.conf`. This block starts TigerVNC, which is a combined VNC/X11 server:

```
~/thunderbird/supervisord.conf
```

```
...
```

```
[program:x11]
priority=0
command=/usr/bin/Xtigervnc -desktop "Thunderbird" -localhost -rfbport 5900 -
SecurityTypes None -AlwaysShared -AcceptKeyEvents -AcceptPointerEvents -
AcceptSetDesktopSize -SendCutText -AcceptCutText :0
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true
```

In this block, you are setting up the X11 server. X11 is a display server protocol, which is what allows GUI applications to run. Note that in the future it will be replaced with Wayland, but remote access is still in development.

For this container, you are using TigerVNC and its built-in VNC server. This has a number of advantages over using a separate X11 and VNC server:

- Faster response time, as the GUI drawing is done directly to the VNC server rather than being done to an intermediary framebuffer (the memory which stores the contents of the screen).
- Automatic screen resizing, which allows the remote application to automatically resize to fit the client (in this case, your web browser window).

If you wish, you can change the argument for the `-desktop` option from `Thunderbird` to something else of your choosing. The server will display your choice as the title of the webpage used to access your application.

Now, let's add a third block of code to `supervisord.conf` to start `easy-novnc`:

~/thunderbird/supervisord.conf

```
...
[program:easy-novnc]
priority=0
command=/usr/local/bin/easy-novnc --addr :8080 --host localhost --port 5900 --no-
url-password --novnc-params "resize=remote"
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true
```

In this block, you are setting up `easy-novnc`, a standalone server which provides a wrapper around noVNC. This server performs two roles. First, it provides a simple connection page which allows you to configure options for the connection, and allows you to set default ones. Second, it proxies VNC over WebSocket, which allows it to be accessed through an ordinary web browser.

Usually, resizing is done on the client side (i.e. image scaling), but you are using the `resize=remote` option to take full advantage of TigerVNC's remote resolution adjustments. This also provides lower latency on slower devices, such as lower-end Chromebooks:

Note: This tutorial uses `easy-novnc`. If you wish, you can use `websockify` and a separate web server instead. The advantage of `easy-novnc` is that the memory usage and startup time is significantly lower and that it's self-contained. `easy-novnc` also provides a cleaner connection page than the default `noVNC` one and allows setting default options that are helpful for this setup (such as `resize=remote`).

Now add the following block to your configuration to start OpenBox, the window manager:

~/thunderbird/supervisord.conf

```
...
[program:openbox]
priority=1
command=/usr/bin/openbox
environment=DISPLAY=:0
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true
```

In this block, you are setting up OpenBox, a lightweight X11 window manager. You could skip this step, but without it, you wouldn't have title bars or be able to resize windows.

Finally, let's add the last block to `supervisord.conf`, which will start the main application:

~/thunderbird/supervisord.conf

```
...
[program:app]
priority=1
environment=DISPLAY=:0
command=/usr/bin/thunderbird
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true
```

In this final block, you are setting `priority` to `1` to ensure that Thunderbird launches after TigerVNC, or it would encounter a race-condition and randomly

fail to start. We also set `autorestart=true` to automatically reopen the application if it mistakenly closes. The `DISPLAY` environment variable tells the application to display on the VNC server you set up earlier.

Here is what your completed `supervisord.conf` will look like:

`~/thunderbird/supervisord.conf`

```
[supervisord]
nodaemon=true
pidfile=/tmp/supervisord.pid
logfile=/dev/fd/1
logfile_maxbytes=0

[program:x11]
priority=0
command=/usr/bin/Xtigervnc -desktop "Thunderbird" -localhost -rfbport 5900 -
SecurityTypes None -AlwaysShared -AcceptKeyEvents -AcceptPointerEvents -
AcceptSetDesktopSize -SendCutText -AcceptCutText :0
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true

[program:easy-novnc]
priority=0
command=/usr/local/bin/easy-novnc --addr :8080 --host localhost --port 5900 --no-
url-password --novnc-params "resize=remote"
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true

[program:openbox]
priority=1
command=/usr/bin/openbox
environment=DISPLAY=:0
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true

[program:app]
priority=1
environment=DISPLAY=:0
command=/usr/bin/thunderbird
autorestart=true
stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
redirect_stderr=true
```

If you want to containerize a different application, replace `/usr/bin/thunderbird` with the path to your application's executable. Otherwise, you are now ready to configure your GUI's main menu.

Step 2 — Setting Up the OpenBox Menu

Now that your process manager is configured, let's set up the OpenBox menu. This menu allows us to launch applications inside the container. We will also include a terminal and process monitor for debugging if required.

Inside your application's directory, use `nano` or your favorite text editor to create and open a new file called `menu.xml`:

```
nano ~/thunderbird/menu.xml
```

Now add the following code to `menu.xml`:

```
~/thunderbird/menu.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<openbox_menu xmlns="http://openbox.org/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://openbox.org/ file:///usr/share/openbox/menu.xsd">
  <menu id="root-menu" label="Openbox 3">
    <item label="Thunderbird">
      <action name="Execute">
        <execute>/usr/bin/thunderbird</execute>
      </action>
    </item>
    <item label="Terminal">
      <action name="Execute">
        <execute>/usr/bin/x-terminal-emulator</execute>
      </action>
    </item>
    <item label="Htop">
      <action name="Execute">
        <execute>/usr/bin/x-terminal-emulator -e htop</execute>
      </action>
    </item>
  </menu>
</openbox_menu>
```

This XML file contains the menu items that will appear when you right-click on the desktop. Each item consists of a label and an action.

If you want to containerize a different application, replace `/usr/bin/thunderbird` with the path to your application's executable and change the `label` of the item.

Step 3 — Creating the Dockerfile

Now that OpenBox is configured, you'll be creating the Dockerfile, which ties everything together.

Create a Dockerfile in your container's directory:

```
nano ~/thunderbird/Dockerfile
```

To begin, let's add some code to build `easy-novnc`:

```
~/thunderbird/Dockerfile
```

```
FROM golang:1.14-buster AS easy-novnc-build
WORKDIR /src
RUN go mod init build && \
    go get github.com/geek1011/easy-novnc@v1.1.0 && \
    go build -o /bin/easy-novnc github.com/geek1011/easy-novnc
```

In the first stage, you are building `easy-novnc`. This is done in a separate stage for simplicity and to save space — you don't need the entire Go toolchain in your final image. Note the `@v1.1.0` in the build command. This ensures that the result is deterministic, which is important because Docker caches the result of each step. If you had not specified an explicit version, Docker would reference the latest version of `easy-novnc` at the time the image was first built. In addition, you want to ensure that you download a specific version of `easy-novnc`, in case breaking changes are made to the CLI interface.

Now let's create the second stage, which will become the final image. Here you will be using Debian 10 (buster) as the base image. Note that since this is running in a container, it will work regardless of the distribution you are running on your server.

Next, add the following block to your Dockerfile:

```
~/thunderbird/Dockerfile
```

```
...
FROM debian:buster
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends openbox tigervnc-standalone-server
supervisor gosu && \
    rm -rf /var/lib/apt/lists && \
    mkdir -p /usr/share/desktop-directories
```

In this instruction, you are installing Debian 10 as your base image and then installing the bare minimum required to run GUI applications in your container. Note that you run `apt-get update` as part of the same instruction to prevent caching issues from Docker. To save space, you are also removing the package lists downloaded afterward (the cached packages themselves are removed by default). You are also creating `/usr/share/desktop-directories` because some applications depend on the directory existing.

Let's add another small block of code:

~/thunderbird/Dockerfile

```
...  
RUN apt-get update -y && \  
    apt-get install -y --no-install-recommends lxterminal nano wget openssh-client  
rsync ca-certificates xdg-utils htop tar xzip gzip bzip2 zip unzip && \  
    rm -rf /var/lib/apt/lists
```

In this instruction, you are installing some useful general-purpose utilities and packages. Of particular interest here are `xdg-utils` (which provides the base commands used by desktop applications on Linux) and `ca-certificates` (which installs the root certificates to allow us to access HTTPS sites).

Now, we can add the instructions for the main application:

~/thunderbird/Dockerfile

```
...  
RUN apt-get update -y && \  
    apt-get install -y --no-install-recommends thunderbird && \  
    rm -rf /var/lib/apt/lists
```

As before, here we are installing the application. If you are containerizing a different application, you can replace these commands with the ones required to install your specific app. Some applications will require a bit more work to run inside Docker. For example, if you are installing an app that uses Chrome, Chromium, or QtWebEngine, you'll need to use the command line argument `--no-sandbox` because it won't be supported within Docker.

Next, let's start adding the instructions to add the last few files to the container:

~/thunderbird/Dockerfile

```
...  
COPY --from=easy-novnc-build /bin/easy-novnc /usr/local/bin/
```



```
COPY menu.xml /etc/xdg/openbox/
COPY supervisord.conf /etc/
EXPOSE 8080
```

Here you are adding the configuration files you created earlier to the image and copying the `easy-novnc` binary from the first stage.

This next code block creates the data directory and adds a dedicated user for your app. This is important because some applications refuse to run as root. It's also good practice not to run applications as root, even in a container.

~/thunderbird/Dockerfile

```
...
RUN groupadd --gid 1000 app && \
    useradd --home-dir /data --shell /bin/bash --uid 1000 --gid 1000 app && \
    mkdir -p /data
VOLUME /data
```

To ensure a consistent `UID/GID` for the files, you are explicitly setting both to `1000`. You are also mounting a volume on the data directory to ensure it persists between restarts.

Finally, let's add the instructions to launch everything:

~/thunderbird/Dockerfile

```
...
CMD ["sh", "-c", "chown app:app /data /dev/stdout && exec gosu app supervisord"]
```

By setting the default command to `supervisord`, the manager will launch the processes required to run your application. In this case, you are using `CMD` rather than `ENTRYPOINT`. In most cases, it wouldn't make a difference, but using `CMD` is better-suited for this purpose for a few reasons. First, `supervisord` doesn't take any arguments that would be relevant to us, and if you provide arguments to the container, they replace `CMD` and are appended to `ENTRYPOINT`. Second, using `CMD` allows us to provide an entirely different command (which will be executed by `/bin/sh -c`) when passing arguments to the container, which makes debugging easier.

And lastly, you need to run `chown` as root before starting `supervisord` to prevent permission issues on the data volume and to allow the child processes to open `stdout`. This also means you need to use `gosu` instead of the `USER` instruction to switch the user.

Here is what your completed `Dockerfile` will look like:

`~/thunderbird/Dockerfile`

```
FROM golang:1.14-buster AS easy-novnc-build
WORKDIR /src
RUN go mod init build && \
    go get github.com/geek1011/easy-novnc@v1.1.0 && \
    go build -o /bin/easy-novnc github.com/geek1011/easy-novnc

FROM debian:buster

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends openbox tigervnc-standalone-server
supervisor gosu && \
    rm -rf /var/lib/apt/lists && \
    mkdir -p /usr/share/desktop-directories

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends lxterminal nano wget openssh-client
rsync ca-certificates xdg-utils htop tar xzip gzip bzip2 zip unzip && \
    rm -rf /var/lib/apt/lists

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends thunderbird && \
    rm -rf /var/lib/apt/lists

COPY --from=easy-novnc-build /bin/easy-novnc /usr/local/bin/
COPY menu.xml /etc/xdg/openbox/
COPY supervisord.conf /etc/
EXPOSE 8080

RUN groupadd --gid 1000 app && \
    useradd --home-dir /data --shell /bin/bash --uid 1000 --gid 1000 app && \
    mkdir -p /data
VOLUME /data

CMD ["sh", "-c", "chown app:app /data /dev/stdout && exec gosu app supervisord"]
```

Save and close your `Dockerfile`. Now we are ready to build and run our container, and then access Thunderbird — a GUI application.

Step 4 — Building and Running the Container

The next step is to build your container and set it to run at startup. You'll also set up a volume to preserve the application data between restarts and updates.

First build your container. Make sure to run these commands in the `~/thunderbird` directory:

```
docker build -t thunderbird .
```

Now create a new network that will be shared between the app's containers:

```
docker network create thunderbird-net
```

Then create a volume to store the application data:

```
docker volume create thunderbird-data
```

Finally, run it and set it to restart automatically:

```
docker run --detach --restart=always --volume=thunderbird-data:/data --  
net=thunderbird-net --name=thunderbird-app thunderbird
```

Note that if you want, you can replace the `thunderbird-app` after the `--name` option with a different name. Whatever you have chosen, your application is now containerized and running. Now let's use the Caddy web server to secure it and remotely connect to it.

Step 5 — Setting up Caddy

In this step, you'll set up the Caddy web server to provide authentication and, optionally, remote file access over WebDAV. For simplicity, and to allow you to use it with your existing reverse proxy, you'll run it in another container.

Create a new directory and then move inside it:

```
mkdir ~/caddy  
cd ~/caddy
```

Now create a new `Dockerfile` using `nano` or your preferred editor:

```
nano ~/caddy/Dockerfile
```

Then add the following directives:

```
~/caddy/Dockerfile
```

```
FROM golang:1.14-buster AS caddy-build  
WORKDIR /src  
RUN echo 'module caddy' > go.mod && \
```

```

    echo 'require github.com/caddyserver/caddy/v2 v2.1.1' >> go.mod && \
    echo 'require github.com/mholt/caddy-webdav v0.0.0-20200523051447-
bc5d19941ac3' >> go.mod
RUN echo 'package main' > caddy.go && \
    echo 'import caddycmd "github.com/caddyserver/caddy/v2/cmd"' >> caddy.go && \
    echo 'import _ "github.com/caddyserver/caddy/v2/modules/standard"' >> caddy.go
&& \
    echo 'import _ "github.com/mholt/caddy-webdav"' >> caddy.go && \
    echo 'func main() { caddycmd.Main() }' >> caddy.go
RUN go build -o /bin/caddy .

FROM debian:buster

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends gosu && \
    rm -rf /var/lib/apt/lists

COPY --from=caddy-build /bin/caddy /usr/local/bin/
COPY Caddyfile /etc/
EXPOSE 8080

RUN groupadd --gid 1000 app && \
    useradd --home-dir /data --shell /bin/bash --uid 1000 --gid 1000 app && \
    mkdir -p /data
VOLUME /data

WORKDIR /data
CMD ["sh", "-c", "chown app:app /data && exec gosu app /usr/local/bin/caddy run -
adapter caddyfile -config /etc/Caddyfile"]

```

This Dockerfile builds Caddy with the WebDAV plugin enabled, and then launches it on port 8080 with the Caddyfile at /etc/Caddyfile. Save and close the file.

Next you will configure the Caddy web server. Create a file named Caddyfile in the directory you just created:

```
nano ~/caddy/Caddyfile
```

Now add the following code block to your Caddyfile:

```
~/caddy/Caddyfile
```

```

{
    order webdav last
}
:8080 {
    log
    root * /data
    reverse_proxy thunderbird-app:8080

    handle_path /files/* {
        file_server browse
    }
    redir /files /files/

    handle /webdav/* {

```

```

    webdav {
        prefix /webdav
    }
}
redir /webdav /webdav/

basicauth /* {
    {env.APP_USERNAME} {env.APP_PASSWORD_HASH}
}
}

```

This Caddyfile proxies the root directory to the `thunderbird-app` container you created in Step 4 (Docker resolves it into the correct IP). It will also serve a read-only web-based file browser on `/files` and run a WebDAV server on `/webdav` which you can mount locally to access your files. The username and password are read from the environment variables `APP_USERNAME` and `APP_PASSWORD_HASH`.

Now build the container:

```
docker build -t thunderbird-caddy .
```

Caddy v.2 requires you to hash your desired password. Run the following command and remember to replace `mypass` with a strong password of your choosing:

```
docker run --rm -it thunderbird-caddy caddy hash-password -plaintext 'mypass'
```

This command will output a string of characters. Copy this to your clipboard in preparation of running the next command.

Now you are ready to run the container. Make sure to replace `myuser` with a username of your choosing, and replace `mypass-hash` with the output of the command you ran in the previous step. You can also change the port (8080 here) to access your server on a different port:

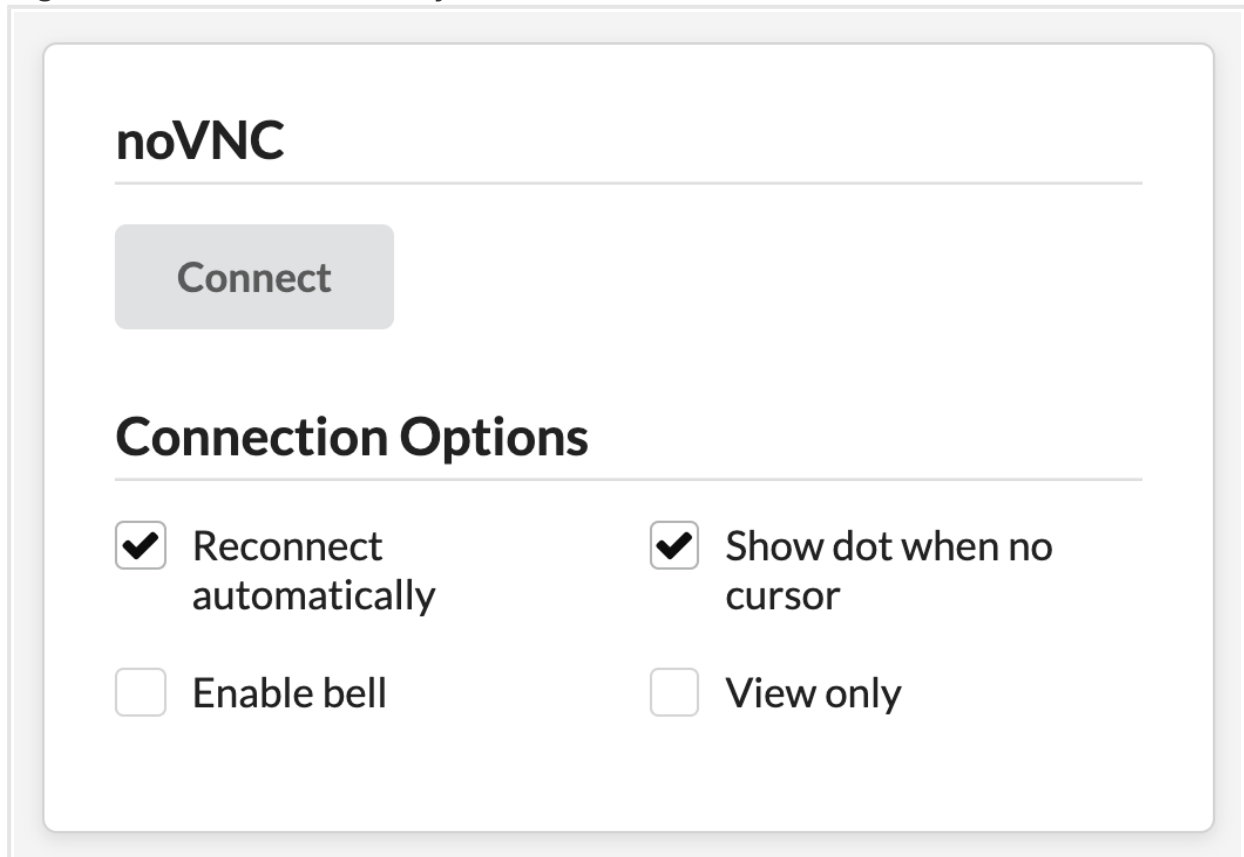
```
docker run --detach --restart=always --volume=thunderbird-data:/data --
net=thunderbird-net --name=thunderbird-web --env=APP_USERNAME="myuser" --
env=APP_PASSWORD_HASH="mypass-hash" --publish=8080:8080 thunderbird-caddy
```

We are now ready to access and test our application.

Step 6 — Testing and Managing the Application

Let's access your application and ensure that it's working.

First, open `http://your_server_ip:8080` in a web browser, log in with the credentials you chose earlier, and click **Connect**.

The image shows a web interface for noVNC. At the top, the text "noVNC" is displayed in a large, bold, black font. Below this, there is a light gray button with the word "Connect" in black text. Underneath the button, the section "Connection Options" is titled in a bold, black font. This section contains four checkboxes arranged in two columns. The first column has "Reconnect automatically" (checked) and "Enable bell" (unchecked). The second column has "Show dot when no cursor" (checked) and "View only" (unchecked).

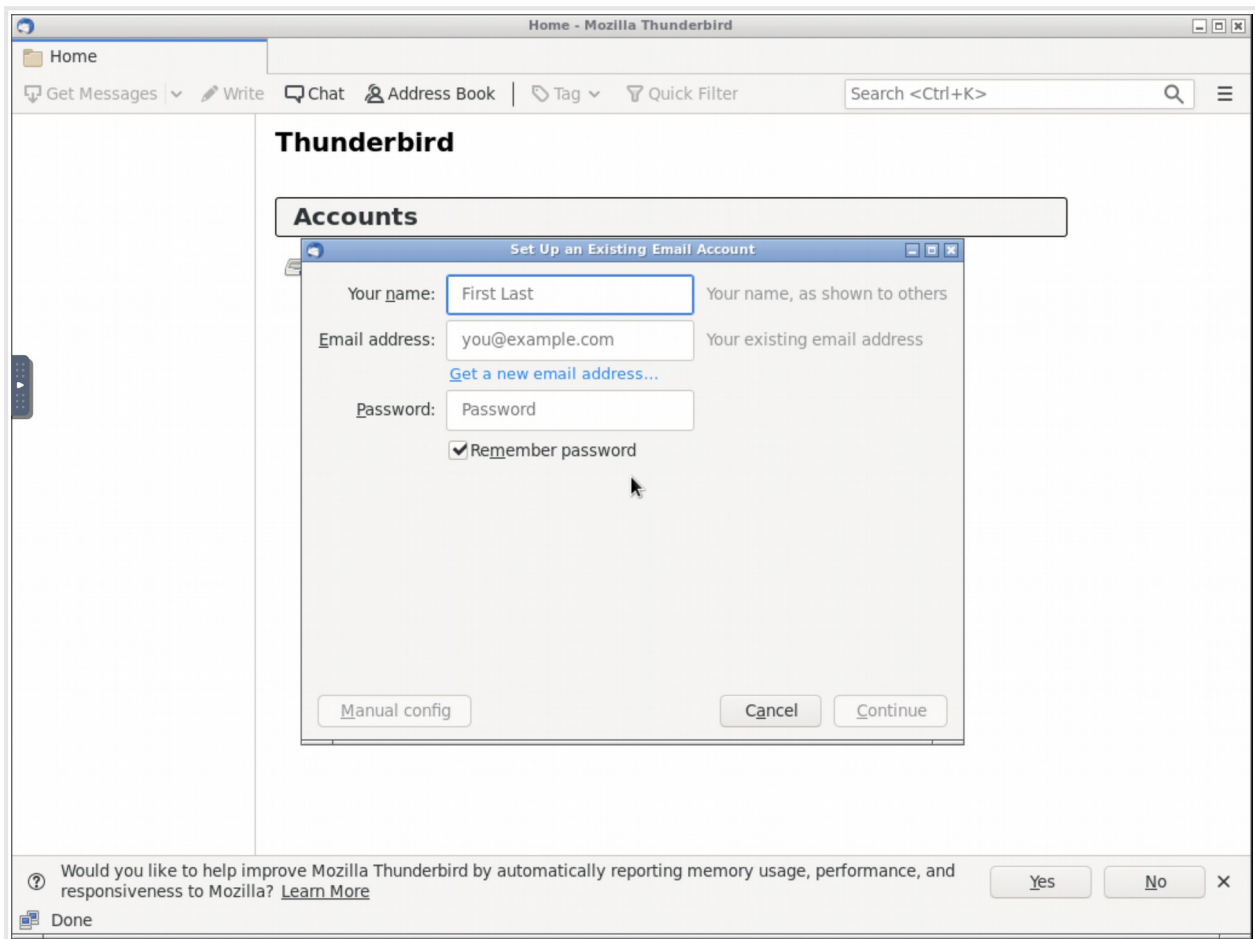
noVNC

Connect

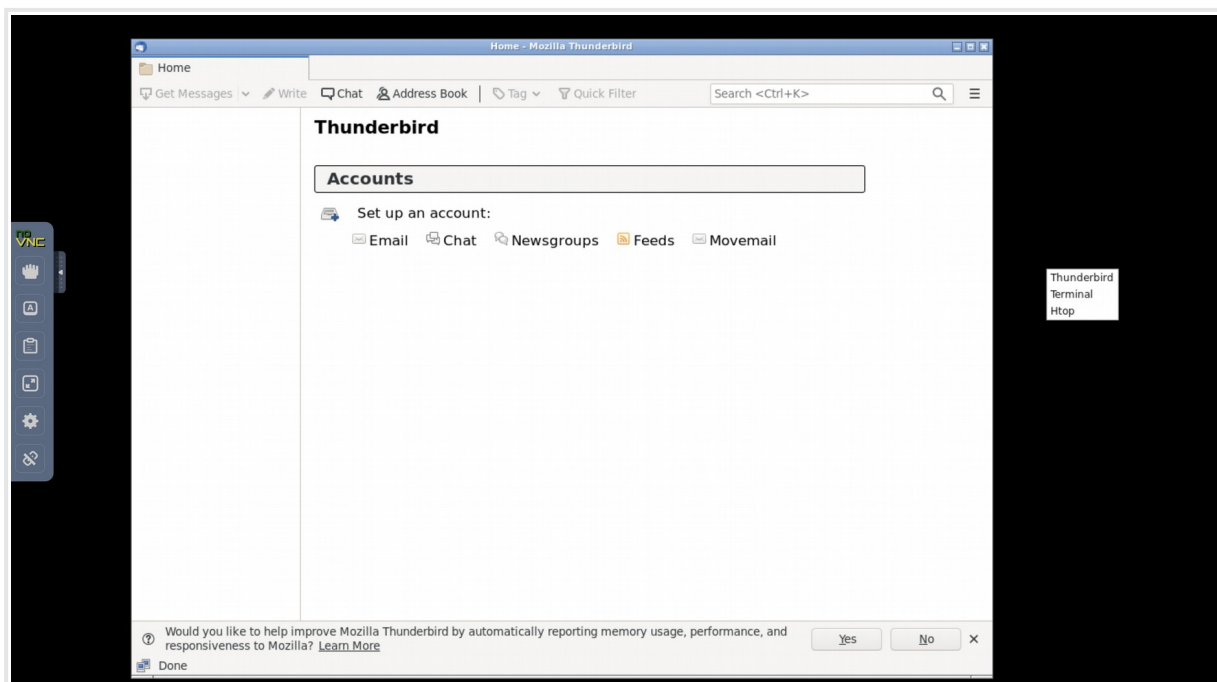
Connection Options

<input checked="" type="checkbox"/> Reconnect automatically	<input checked="" type="checkbox"/> Show dot when no cursor
<input type="checkbox"/> Enable bell	<input type="checkbox"/> View only

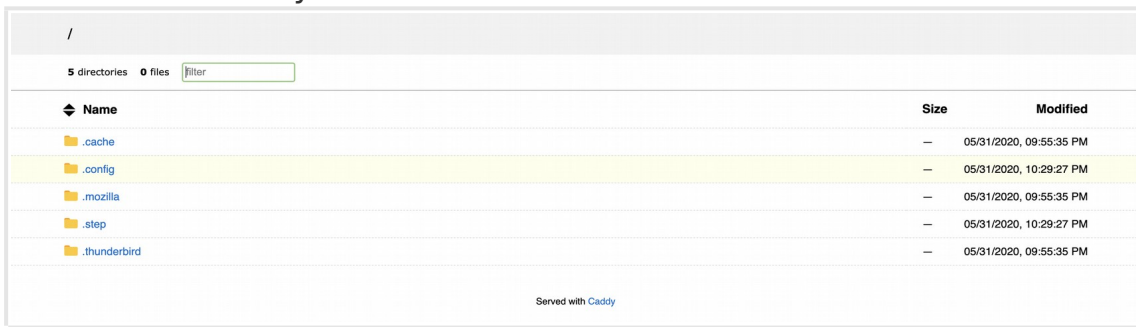
You should now be able to interact with the application, and it should automatically resize to fit your browser window.



If you right-click on the black desktop, you should see a menu that allows you to access a terminal. If you middle-click, you should see a list of windows.



Now open `http://your_server_ip:8080/files/` in a web browser. You should be able to access your files.



Name	Size	Modified
.cache	—	05/31/2020, 09:55:35 PM
.config	—	05/31/2020, 10:29:27 PM
.mozilla	—	05/31/2020, 09:55:35 PM
.step	—	05/31/2020, 10:29:27 PM
.thunderbird	—	05/31/2020, 09:55:35 PM

Optionally, you can try mounting `http://your_server_ip:8080/webdav/` in a WebDAV client. You should be able to access and modify your files directly. If you use the **Map network drive** option in Windows Explorer, you will either need to use a reverse proxy to add HTTPS or set `HKLM\SYSTEM\CurrentControlSet\Services\WebClient\Parameters\BasicAuthLevel` to `WORD:2`.

In either case, your native GUI application is now ready for remote use.

Conclusion

You have now successfully set up a Docker container for Thunderbird and then, using Caddy, you've configured access to it through a web browser. Should you ever need to upgrade your app, stop the containers, run `docker rm thunderbird-app thunderbird-web`, re-build the images, and then re-run the `docker run` commands from the previous steps above. Your data will still be preserved since it is stored in a volume.

If you want to learn more about basic Docker commands, you can read [this tutorial](#) or [this cheatsheet](#). For longer-term use, you may also want to consider enabling HTTPS (this requires a domain) for additional security.

Additionally, if you're deploying more than one application, you may want to use Docker Compose or Kubernetes instead of starting each container manually. And remember, this tutorial can serve as a base for running any other Linux application on your server, including:

- [Wine](#), a compatibility layer for running Windows applications on Linux.
- [GIMP](#), an open-source image editor.

- Cutter, an open-source reverse engineering platform.

This last option demonstrates the great potential of containerizing and remotely accessing GUI applications. With this setup, you can now use a server with considerably more computing power than you might have locally to run resource-intensive tools like Cutter.