SELF HOSTED 101

The Zero-Jargon Guide for Absolute Beginners

Self-Hosted 101

The Zero-Jargon Guide for Absolute Beginners

A Practical Guide to Hosting Your Own Services

By a Fellow Beginner

Disclaimer

This guide is for educational purposes. Always backup your data before making changes to your system. The author is not responsible for any data loss or system damage.

Introduction

Welcome to Self-Hosted 101! This guide will take you from zero knowledge to running your own services at home. No technical background required.

By the end of this guide, you'll have: - A working self-hosted server - Multiple useful services running - The knowledge to expand further

Let's begin your journey!

Note: This guide was created in February 2026

Chapter 1: Why Bother? (The "Why" Before the "How")

This chapter is about setting realistic expectations. We'll cover what self-hosting actually solves, what it doesn't solve, and whether it's right for you. No technical stuff here–just honest conversation about whether this journey is worth your time.

So... Why Are You Here?

Maybe you saw a viral Reddit post about someone who self-hosted everything and thought "I want that." Maybe you're tired of paying monthly subscriptions for services you barely use. Maybe you just want more control over your digital life. Or maybe–and this is totally valid–you're just curious.

None of those reasons are wrong. But before we dive into the how, let's talk about the why. Because self-hosting isn't for everyone, and I want you to finish this chapter knowing exactly what you're signing up for.

What Self-Hosting Actually Means

Let's start with a simple definition. Self-hosting means running your own services–your own email, your own media server, your own password manager, your own website–on hardware you own and control, instead of relying on companies like Google, Netflix, or Dropbox to do it for you.

Think of it like cooking at home versus always eating out. When you eat out, someone else decides the ingredients, controls the environment, and can change the menu whenever they want. When you cook at home, you pick what goes in, you control the quality, and you can adjust recipes to your taste. It's more work, but you own it.

That's self-hosting in a nutshell.

What Self-Hosting Can Do For You

Here's the good stuff–the reasons thousands of people (myself included) spend time on this:

Privacy. When you host your own data, it's yours. No company scanning your photos for ad targeting. No algorithm learning your viewing habits. Your passwords, your files, your emails–they stay on your hardware, under your roof. For many people, this alone is worth the effort.

Control. Companies change policies. Services shut down. Remember Google+? Remember Vine? When you rely on third-party services, you're always one corporate decision away from losing access to something you depend on. When

you host yourself, you decide when to update, when to upgrade, and when to switch things around.

Learning. This one surprises people, but many folks get into self-hosting just to learn. You'll pick up Linux, networking, security basics–the kind of skills that open doors and change how you understand technology. Even if you never become a "tech person," you'll understand how the internet actually works under the hood.

Saving money (sometimes). Yes, you can self-host alternatives to paid services. A Jellyfin server can replace Netflix. A Vaultwarden instance can replace LastPass. A Nextcloud setup can replace Dropbox. But–and this is important–the savings usually come from the value of your time, not just the subscription fees. We'll talk more about this later.

Customization. Want your photo gallery to look a certain way? Want your media server to organize your files in a specific structure? Want to combine services in ways no one else has thought of? When you host yourself, you're not limited by what a company decided to build.

What Self-Hosting Can't Do

Now for the reality check. Self-hosting isn't magic, and it's not for everyone. Here's what it won't do:

It won't make everything free. You still need to pay for electricity, internet, hardware, and occasionally domain names or cloud services. The costs are different, not zero.

It won't be as polished as big tech products. Netflix has thousands of engineers making their interface perfect. You have yourself (and maybe the open-source community). That's powerful, but the user experience won't always be as slick.

It won't automatically be more private if you don't configure it right. Self-hosting doesn't mean "secure by default." You'll need to set things up properly, which we'll cover. A poorly configured self-hosted service can actually be less secure than a well-managed cloud service.

It won't run forever without attention. This is the big one, and I'm putting it in bold because it's the most common mistake: **self-hosting requires maintenance.** We'll talk about this more in a moment.

The "Set and Forget" Myth

I need to address something you'll see all over the internet: the idea that you can "set it and forget it."

You can't.

Here's what actually happens. You install a service, everything works great, and then... you mostly forget about it. That's fine for a while. But eventually, updates come out. Security vulnerabilities are discovered. Your hardware might have issues. Your internet might change. Services evolve.

If you never check on your servers, problems will pile up. Outdated software becomes a security risk. Failed hard drives can wipe out data. Forgotten configurations can cause unexpected issues.

This doesn't mean you need to micromanage everything. It means self-hosting is more like owning a car than renting a streaming service. You can drive it every day without thinking about it, but you still need an oil change occasionally and you'll probably need to replace the tires eventually.

The good news? Once you get things running, the maintenance isn't that bad. Maybe an hour or two a month once you're established. We'll cover this in detail in Chapter 10.

What Kind of Person Is Self-Hosting For?

Let me be honest. Self-hosting is worth it if:

- You're curious about how things work
- You have a specific problem you want to solve (like ad blocking across your whole network, or having a personal media server)
- You're okay with spending some time learning
- You value control and ownership over convenience
- You don't mind troubleshooting when things go wrong

Self-hosting might not be worth it if:

- You just want things to work and don't care how
- You have no time to learn new things
- You need 24/7 professional support when something breaks
- You're expecting to save a lot of money (the time investment rarely adds up financially)

Neither answer is wrong. It's just different priorities.

Time Commitment: Setting Realistic Expectations

Let's talk numbers. When you're starting out, expect to spend:

- **Chapter 2-3 (Hardware & Setup):** A few hours researching, maybe a weekend to get your first machine running
- **Chapter 4-5 (First Services):** 1-2 hours per service to get comfortable
- **Chapter 6-8 (Networking):** A few hours to understand how to access things remotely
- **Chapter 9-10 (Security & Maintenance):** Ongoing, but gets faster as you learn

After that initial learning curve (maybe 2-4 weeks if you go at a relaxed pace), maintaining your setup might take:

- **15-30 minutes a week** for basic monitoring and updates
- **A few hours a month** for more thorough checks and backups

This is assuming you start simple and grow gradually. If you try to host twenty services on day one, you'll be overwhelmed. We'll make sure you don't do that.

What Do You Want to Solve?

Before we move on, let's figure out your "why." Self-hosting isn't one thing–it's a whole world of possibilities. Here are the most common reasons people start:

Ad blocking network-wide. Imagine blocking ads on your phone, your laptop, your smart TV, and your tablet all at once, without installing anything on any of them. AdGuard Home does this. It's the easiest, most rewarding first project, and we'll do it in Chapter 5.

Personal media server. Your own Netflix, but for videos you've downloaded or ripped from your own discs. Your own Spotify, but for music you've collected. Jellyfin handles this, and it's incredibly satisfying.

Password management. LastPass got breached. 1Password keeps raising prices. Vaultwarden gives you a professional-grade password manager that you own, for free.

File sync and backup. Stop trusting your files to companies. Set up your own cloud–Nextcloud or Immich let you sync photos, files, and more across all your devices.

Smart home. Home Assistant is massive–thousands of devices, endless possibilities. It's a whole hobby on its own.

Home automation. Connect services together. Automatically download things, organize files, send notifications. n8n and other tools let you build workflows that would cost a fortune if bought as a service.

Learning. Honestly, many people start "just to learn" and end up with a setup they use every day. That's perfectly valid too.

The Path Ahead

Here's how this guide works. We're going to build up slowly:

1. **First**, you'll figure out what hardware to use (Chapter 2)
2. **Then**, you'll set up your first machine (Chapter 3)
3. **Then**, we'll explain the magic of containers (Chapter 4)
4. **Then**, you'll install your first service (Chapter 5)
5. **Then**, we'll make sense of networking (Chapter 6)
6. **And so on**, adding more services and capabilities as you go

Each chapter builds on the last. By the end, you'll have a working self-hosted setup, and more importantly, you'll understand what you've built.

Do This Now ?

Grab a piece of paper or open a note on your phone. Write down **three things** you want to achieve by self-hosting.

Not what you think you "should" want. What you actually want.

Maybe it's "block ads on my phone." Maybe it's "have my own Netflix." Maybe it's "never pay for LastPass again." Maybe it's "learn how servers work."

Write them down. Keep them somewhere you'll see them later. When things get technical (and they will), these three reasons will remind you why you started.

What's Next?

Ready? Let's talk hardware. Chapter 2 is where we figure out what machine you'll use. I'll show you why most advice about Raspberry Pis is outdated, and why a cheap used desktop might actually be your best option.

Let's go.

Chapter 2: What Do You Actually Need?

The Hardware Question

Here's the good news: you probably already have everything you need to get started.

Before we dive in, let's clear up something important. You might have heard people raving about Raspberry Pi computers. They're small, cheap, and look cool on a desk. But here's the truth: for most beginners, a Raspberry Pi is actually the *wrong* choice.

Why? Two reasons. First, many self-hosted software tools don't work well on Raspberry Pi. Developers often skip optimizing for it, which means you might find a cool service you want to run, only to discover it won't run on your Pi. Second, a Raspberry Pi is slower than you'd expect. That $50 computer can struggle when you're running more than a couple of services.

So what should you use instead?

The Sweet Spot: Used Desktop or NUC

The best choice for most beginners is a used desktop computer or a small computer called a NUC (which just means "Next Unit of Computing"). These are everywhere, affordable, and powerful enough to run everything you'll want.

Think about it: every year, companies upgrade their office computers. Those old machines still work perfectly fine–they're just not the newest anymore. You can often pick one up for $50-150, and it'll run circles around a Raspberry Pi.

Understanding the Specs (Plain English)

When you're looking at computers, you'll see numbers and terms that feel confusing. Let's make sense of them:

Processor (CPU): This is the brain of the computer. You want something from the last 5-7 years. If you see "Intel Core i3," "Intel Core i5," or "AMD Ryzen 3/5," you're in good shape. Don't worry about the exact model–anything from this decade will work fine.

Memory (RAM): This is short-term memory–how many things the computer can think about at once. Think of it like a desk: bigger desk, more room to work. You want at least 4 gigabytes (4GB). 8GB is better and gives you room to grow. If you see "8GB RAM," that's perfect.

Storage (Hard Drive/SSD): This is long-term memory–where your files and programs live. You don't need much to start. 128 gigabytes (128GB) is enough for learning. 256GB or 500GB is better if you plan to store photos or videos. A SSD (solid state drive) is faster than a regular hard drive–get one if you can, but it's not required.

The x86 vs ARM Thing (Simplified)

You might hear people talk about "x86" and "ARM." Here's what that means in plain English:

x86 is the type of processor in most regular computers. It's powerful and runs almost all software without problems. Every Intel or AMD processor you've ever owned is x86.

ARM is the type of processor in phones, tablets, and Raspberry Pis. It's designed to be energy-efficient and run cooler. But many software tools are written for x86 and don't work well on ARM.

The takeaway: get a computer with an Intel or AMD processor (x86). It'll run everything you want to try.

What to Look For

Here's your shopping checklist:

? **Processor:** Intel Core i3/i5/i7 or AMD Ryzen 3/5/7 (from the last few years) ? **Memory:** At least 4GB, ideally 8GB ? **Storage:** At least 128GB (SSD preferred, but HDD works) ? **Size:** Small desktop or NUC preferred (quiet and takes less space) ? **Condition:** Works fine–doesn't need to look pretty

What to avoid: ? Raspberry Pi (for now–we'll explain when it makes sense later) ? Computers older than 10 years ? Anything with less than 2GB RAM

Where to Find Hardware

Facebook Marketplace: Search your local area for "desktop computer" or "used PC." People are constantly selling office upgrades. You can often find good deals and inspect the machine before buying.

eBay: Great for comparison shopping. Look for "refurbished" or "used - good condition" desktops. Check the specs carefully and read the description.

Craigslist: Similar to Facebook Marketplace. Meet locally and test the machine if possible.

Local computer shops: Some cities have small shops that repair and resell computers. You might pay a bit more, but you get to ask questions and sometimes get a short warranty.

Friends and family: Ask around! Someone you know probably has an old computer gathering dust. They might even give it to you.

Community Recommendations

The self-hosting community has pretty solid consensus on this topic. Here's what experienced self-hosters recommend:

Budget option ($0-50): Use what you have. If you have an old laptop or desktop from the last 8 years, it probably works fine. Try it first before buying anything.

Sweet spot ($50-150): A used small desktop with 8GB RAM. Look for Dell OptiPlex, HP ProDesk, or Lenovo ThinkCentre. These business machines were built to run all day, every day–and they last.

If you want new ($200-300): A Intel NUC or similar small computer. These are designed to be quiet and efficient. Great if you want something that looks nice in your home.

The Real Cost

Let's be honest about money:

- **Used desktop:** $50-150
- **Electricity:** Maybe $5-15 per month (depends on your electricity rates and how much you run it)
- **Internet:** You already have this

That's it. Unlike cloud services that charge you monthly forever, your self-hosted server is a one-time purchase that keeps on giving.

Do This Now

Go look around your home. Check that closet, that corner, the old desk in the garage.

Do you have an old computer sitting somewhere? Laptop or desktop, it doesn't matter. Turn it on and see if it still works.

If you find one that boots up–you're done. You have your server.

If you don't find anything, make a note: "Need to find a used desktop." Then check Facebook Marketplace or eBay this week. Set a budget of $100 and see what's available in your area.

You don't need to buy anything today. Just know what's out there.

What's Next

Once you have your hardware sorted, we'll get you set up with the software. Chapter 3 walks you through installing Ubuntu Server–the operating system that runs your self-hosted world. No prior experience needed. We'll make it easy.

Chapter 3: Setting Up Your First Machine

Estimated reading time: 15 minutes Estimated "do this now" time: 20 minutes

---

In the last chapter, you figured out what hardware to use. Now it's time to turn that old computer into something useful. We're going to install **Ubuntu Server** – a free operating system that turns your machine into a proper server.

Don't worry if that sentence made no sense. We're about to break it all down.

What Even Is Ubuntu Server?

Let's start with some definitions, because this chapter is going to throw some new words at you.

Operating System (or OS) is the main software that runs your computer. It's the thing that lets you move your mouse, open apps, and save files. Windows and macOS are operating systems. Ubuntu Server is another one – but instead of giving you a desktop with icons and windows, it gives you a *text-only* interface. No pictures, no clicking. Just words on a screen.

Think of it like the difference between a car's dashboard (with screens and buttons) and a race car cockpit (with just essential controls). Ubuntu Server is stripped down and designed to run reliably without any extra bells and whistles.

Ubuntu (pronounced "oo-BOON-too") is a flavor of Linux. Linux is an operating system – the same kind that runs most of the internet, your Android phone, and supercomputers. Ubuntu is the most beginner-friendly version of Linux. It's free, it's secure, and it has a huge community to help you.

Server doesn't mean a fancy piece of hardware. It just means "a computer that provides services to other computers." Your server will provide services like ad-blocking, media streaming, and password storage to the other devices in your home.

So when we say "Ubuntu Server," we mean: a free, reliable operating system that runs without a graphical interface, designed to serve other devices on your network.

Simple enough? Let's move on.

Why Ubuntu Server?

You might be wondering: why not just use Windows?

Three reasons:

1. **It's free.** No license fees, ever.
2. **It's stable.** Servers run for months or years without needing to be restarted.

3. **It's secure.** Linux has far fewer viruses and malware than Windows.

Ubuntu Server also makes installing software incredibly easy – which is exactly what we'll do in the next chapters.

What You'll Need Before We Start

Before we begin the installation, gather these things:

- **Your target computer** (the one you want to turn into a server)
- **Another computer** (laptop or desktop) to create the installer
- **A USB stick** (at least 8GB – this will be erased, so use one you don't need)
- **Internet connection** (for downloading Ubuntu and updates)
- **About 30 minutes** of uninterrupted time

Step 1: Download Ubuntu Server

On your *other* computer (not the server), open your web browser and go to:

https://ubuntu.com/download/server

You should see a big button that says "Download Ubuntu Server." Click it.

Important: Make sure you download the **Server** version, not the Desktop version. Desktop has pictures and windows; Server is text-only. We want Server.

The file you download will be called something like `ubuntu-24.04-live-server-amd64.iso`. The `.iso` part just means "image file" – it's a complete copy of Ubuntu packaged into one file.

This download might take a few minutes depending on your internet speed. While it's downloading, let's prepare the USB stick.

Step 2: Make a Bootable USB

An ISO file isn't something you can just "open." To use it, we need to turn it into a **bootable USB** – a USB stick that can start your computer and run Ubuntu.

What Does "Bootable" Mean?

When you turn on your computer, it follows a set of instructions to get started. This process is called **booting**. Normally, your computer boots from its hard drive into Windows or macOS.

A bootable USB tells your computer: "Instead of starting from your hard drive, start from this USB stick instead." It's like inserting a different instruction manual for how to start up.

How to Create One

The easiest way is using a free tool called **Rufus** (Windows) or **Etcher** (Mac/Windows). Here's how:

Using Etcher (works on Mac and Windows):

1. Go to https://etcher.balena.io/ and download Etcher
2. Install and open it
3. Click "Flash from file" and select your Ubuntu `.iso` file
4. Click "Select target" and choose your USB stick
5. Click "Flash!" and wait for it to finish

Using Rufus (Windows):

1. Go to https://rufus.ie/ and download Rufus
2. Open it (no installation needed)
3. Under "Device," select your USB stick
4. Click "Select" and choose your Ubuntu `.iso` file
5. Leave everything else as default
6. Click "Start" and wait for it to finish

> ? **Warning:** This will completely erase everything on the USB stick. Make sure you've saved anything important from it first.

When it's done, you have a Ubuntu installer on a USB stick. Pretty cool, right?

Step 3: Prepare Your Server Computer

Now let's get your target computer ready. Here's what to do:

1. **Turn off the computer** completely
2. **Plug in the USB stick** into a USB port
3. **Turn on the computer** and watch the screen carefully

You'll see a brief message that says something like "Press F2 to enter setup" or "Press DEL for BIOS." This is your cue to tap a key (usually F2, F12, DELETE, or ESC) to open the **BIOS** – the basic software that controls your computer's hardware.

> **What is the BIOS?** Think of it as the control center for your computer's fundamental settings. It's where you decide which devices to boot from. We need to tell it to boot from our USB stick first.

Once you're in the BIOS:

1. Look for a "Boot" or "Boot Order" section
2. Move your USB stick to the top of the list
3. Save and exit (usually by pressing F10 and confirming)

Your computer should now restart and boot from the USB stick.

Step 4: The Installation Process

When your computer boots from the USB, you'll see a purple screen with the Ubuntu logo. After a moment, you'll see a text-based menu. Here's what to do:

Screen 1: Welcome

You'll see options to try Ubuntu or install it. Press **Enter** on "Install Ubuntu Server."

Screen 2: Keyboard Layout

Use the arrow keys to select your keyboard layout. Most people can just press **Enter** to accept the default (English (US)). If you use a different layout, find it in the list.

Screen 3: Type of Installation

You'll see options like "Ubuntu Server" and "Ubuntu Server (minimized)." Select "Ubuntu Server" and press **Enter**.

Screen 4: Network Configuration

Ubuntu will try to automatically detect your network. If you're using Ethernet (plugging into your router with a cable), it should just work. You'll see it list your network connection.

If you're using Wi-Fi, you'll need to select your network and enter your password. Use the arrow keys to navigate, Tab to move between fields, and Enter to select.

> **What is Ethernet?** It's the standard way to connect computers to the internet using a cable. It looks like a wider phone cable with clips on the side. Most servers use Ethernet because it's faster and more reliable than Wi-Fi.

Screen 5: Storage Configuration

This is the part where we decide how Ubuntu uses your hard drive. For a beginner, the easiest option is to use the entire disk.

1. Select "Use an entire disk"
2. Select the disk you want to use (usually there's only one option)
3. Review the changes – Ubuntu will show you what it's going to do
4. Select "Done" and confirm

> ? **Warning:** This will erase everything on the selected hard drive. Make sure you've backed up any data you wanted to keep.

Screen 6: Your Profile

Now Ubuntu needs to know who you are. You'll set up:

- **Your name** – your full name
- **Server's name** – a name for this computer (like "myserver" or "home-server")
- **Username** – the name you'll use to log in (pick something short, like "admin" or your name)
- **Password** – pick a password you'll remember

  What is a username? It's the name you type to identify yourself to the computer. On personal computers, you might log in as "John" or "Mom." Here, you'll create a username for yourself.

  What is a password? The secret phrase that proves you are who you say you are. Choose something you'll remember!

Screen 7: SSH Setup

SSH (pronounced "S-S-H") stands for **Secure Shell**. It's a way to connect to your server from another computer using text commands. We'll use this in just a few minutes.

When asked if you want to "Install OpenSSH server," select **Yes**. This makes it easier to control your server remotely.

Screen 8: Server Snaps

Ubuntu will ask what software you'd like to install. For now, just press **Enter** to continue without selecting anything extra. We'll install software ourselves in later chapters.

The Final Wait

Now Ubuntu installs. This takes 5-10 minutes. You'll see a progress bar. Go refill your water bottle – you've earned it.

When it's done, you'll see a message saying "Installation complete!" Remove the USB stick and press **Enter** to restart.

Step 5: Log In For the First Time

Your computer restarts. This time, instead of Windows or the USB menu, you'll see a black screen with text. It looks like this:

```
Ubuntu 24.04 LTS myserver tty1
myserver login: _
```

This is called the **terminal** or **command line**. Instead of clicking icons, you type commands. It's how people used to use all computers – and it's still the most powerful way to use a server.

> **What is the terminal?** It's a text-only interface to your computer. Instead of pointing and clicking, you type instructions. The cursor (the blinking line) is where you type.
>
> **What is a command?** An instruction you type for the computer to follow. For example, typing `ls` and pressing Enter tells the computer to "list files."

To log in, type the username you created (press Enter) and then type your password (press Enter).

> **Note:** When typing your password, nothing appears on screen. This is normal – it's a security feature so nobody can see how many characters your password has. Just type it and press Enter.

If you typed everything correctly, you'll see a welcome message and a prompt that looks like:

```
username@myserver:~$ _
```

Congratulations! You're now logged into your server.

Understanding What You're Seeing

Let me break down that prompt:

```
username@myserver:~$
```

- **username** – that's you (your login name)
- **@** – "at" (you are at)
- **myserver** – the name of this computer
- ~ – your home folder (where your personal files go)
- **$** – means you're a regular user (not the administrator)

> **What is a folder?** A place where you store files. It's the same thing as a "directory." Your documents, photos, and downloads are all in folders.
>
> **What is the home folder?** The personal space for your user account. When you log in, this is where you "start."
>
> **What does $ mean?** In the terminal, `$` means you're a normal user with limited permissions. If you see `#`, you're the administrator (root user). We'll get to why that matters later.

Looking Around: Your First Commands

Let's try a few simple commands. Don't worry – you can't break anything by typing these.

Command 1: See Where You Are

Type this and press Enter:

```
pwd
```

This stands for "print working directory." It shows you what folder you're currently in. You should see `/home/yourusername` – that's your home folder.

Command 2: List Files

Type this and press Enter:

```
ls
```

This lists the files in your current folder. Right now, it's probably empty – that's fine.

Command 3: See Who You Are

Type this and press Enter:

```
whoami
```

This tells you what username you're logged in as.

Command 4: Check the Date

Type this and press Enter:

```
date
```

Just so you know, your server doesn't have a fancy calendar app – you have to ask it politely with this command.

Command 5: Get Help

Type this and press Enter:

```
help
```

This shows a list of available commands. Don't worry about memorizing them – we'll learn the ones we need as we go.

The Linux Directory Structure

Now that you're logged in, let's talk about how Linux organizes files. Understanding this now will save you confusion later.

The Root Folder

In Linux, everything starts from a single folder called the **root** folder. Think of it as the trunk of a tree – all other folders branch out from it.

The root folder is written as `/` (just a forward slash).

Important Folders

Here are the folders you'll encounter most:

- `/` – The root (the main starting point)
- `/home` – Where user files live (like `/home/yourusername`)
- `/etc` – Configuration files (settings for programs)
- `/var` – Variable data (logs, databases)
- `/opt` – Optional/extra software
- `/root` – The administrator's home folder (yes, it's different!)
- `/tmp` – Temporary files (these get deleted when you restart)

  What is a configuration file? A special file that tells a program how to behave. It's like settings – you change them to customize how the software works.

Don't worry if this doesn't fully make sense yet. You'll see these folders in action as we install services.

Shutting Down (Safely!)

When you want to turn off your server, don't just hold the power button. That could corrupt files.

Instead, use this command:

```
sudo shutdown now
```

> **What is sudo?** It's a command that means "do this as the administrator." You'll need it for any command that affects the system. It will ask for your password.

Your server will shut down gracefully. When the screen goes black, you can safely unplug it.

? Do This Now

This is your hands-on exercise for this chapter:

Install Ubuntu Server on your hardware

Here's what to do:

1. Download Ubuntu Server from https://ubuntu.com/download/server
2. Create a bootable USB using Etcher or Rufus

3. Boot your server from the USB (remember: press F2/DEL/F12 to enter BIOS)
4. Follow the installation steps in this chapter
5. Log in and run these commands:
   - `pwd` – to see where you are
   - `ls` – to list files
   - `whoami` – to confirm your username
   - `date` – to check the time

Take a screenshot of your terminal after logging in. You did it!

What Comes Next?

You now have a running Ubuntu Server. It's doing nothing useful yet – but that's about to change.

In the next chapter, we're going to install **Docker** – a tool that makes installing and running software incredibly easy. Docker is like having standardized shipping containers for apps: they arrive ready to go, they don't interfere with each other, and you can move them anywhere.

See you there.

Chapter Summary:

- Ubuntu Server is a free, stable, secure operating system for servers
- A bootable USB lets you install Ubuntu from a USB stick
- The BIOS controls how your computer starts up
- The terminal is a text-based interface where you type commands
- Linux organizes files in a tree structure starting from `/` (root)
- Always use `sudo shutdown now` to turn off your server safely

Chapter 4: Docker - Your First Container

Welcome to the World of Containers

Remember when we talked about shipping containers in Chapter 1? Those standardized metal boxes that revolutionized global trade? Well, get ready because Docker does exactly the same thing for software.

Think about how shipping works. Before containers came along, loading a ship was a nightmare. Workers had to load each item individually–crates of different sizes, fragile goods, heavy machinery. It took forever, things got damaged, and nobody could agree on how to pack it all.

Then someone invented the shipping container. Standard size. Standard way of loading. You put your stuff in, seal it up, and it works everywhere. The truck, the train, the ship–they all handle the same container the same way. Game changer.

Docker is like that for software. Instead of worrying about whether your program will run on someone else's computer, you package it up in a standardized "container" that works the same way everywhere.

Why Does This Matter?

Here's the problem Docker solves: remember when you installed that program on your friend's computer and it didn't work? Maybe they had a different version of Python. Or their operating system was different. Or some library was missing. Classic "works on my machine" problem.

With Docker, your software travels in its own little box that includes: - The program itself - Everything the program needs to run - Its own mini-operating system

Nothing from the outside can mess with it. And nothing from inside can mess with your actual computer. It's completely isolated. Pretty cool, right?

The Language of Containers

Before we dive in, let's learn a few terms. I promise they're simple:

Image - Think of this as the blueprint or the recipe. It's the template that tells Docker how to build your container. You don't run an image directly; you use it to create containers.

Container - This is the actual running version of the image. It's like using your recipe to bake a cake–the recipe is the image, the cake is the container.

Volume - Remember when we talked about saving your files in Chapter 3? A volume is Docker's way of saying "keep this data safe even if the container gets

deleted." It's like a separate compartment that survives when you throw away the container.

Dockerfile - A text file that tells Docker how to build your image. Like writing out a recipe step by step.

That's it. Four words. Image, container, volume, Dockerfile. You've got this.

Installing Docker

Now let's get Docker installed on your server. We're going to use the official install script because it handles all the tricky parts for us.

Step 1: Download and Run the Install Script

Open your terminal (or SSH into your server like we learned in Chapter 3) and type this:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

What does this do? - `curl` downloads a script from the internet - `sudo` runs it with special permissions (admin rights) - The script sets up Docker on your system

You'll see a lot of text scroll by. That's normal–it's just the script doing its job.

Step 2: Add Your User to the Docker Group

By default, only special administrators can use Docker. Let's add your user so you don't have to type passwords every time:

```
sudo usermod -aG docker $USER
```

After this command, you'll need to log out and log back in for the change to take effect. If you're SSH'd in, just reconnect.

Step 3: Verify It Works

Let's make sure Docker installed correctly:

```
docker --version
```

You should see something like `Docker version 24.0.0` or similar. If you do, congratulations! Docker is installed.

Running Your First Container

This is the moment you've been waiting for. Let's run the classic "hello world" of containers:

```
docker run hello-world
```

Watch what happens. Docker will: 1. Look for a "hello-world" image on your computer 2. Not find it (because you just installed Docker) 3. Automatically download it from the internet (Docker Hub, which is like an app store for container images) 4. Run it in a container 5. Show you a friendly message

The output should look something like this:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

If you see this message, give yourself a pat on the back. You've just run your first container!

What's Actually Happening Here?

Let me break down what `docker run hello-world` does, piece by piece:

- `docker` - You're talking to the Docker program
- `run` - "I want to start a container"
- `hello-world` - "Use this image to create the container"

The magic is that Docker figured out you didn't have the image locally, so it automatically downloaded it. Next time you run it, it'll be much faster because the image is already on your computer.

Looking at Your Containers

Let's see what containers you have running:

```
docker ps
```

This shows "running" containers. Right now, you probably don't have any running because hello-world just says hi and exits.

Let's see ALL containers (including ones that finished):

```
docker ps -a
```

Now you'll see the hello-world container in the list. It probably shows "Exited" because it finished its job.

Cleaning Up

Those exited containers take up a little space. Let's remove it:

```
docker rm $(docker ps -aq)
```

This removes all stopped containers. Don't worry–you can always run hello-world again anytime.

A More Interesting Example

Hello-world is fun, but let's run something actually useful. Let's run a simple web server that just displays a webpage:

```
docker run -d -p 8080:80 nginx
```

Whoa, there's a lot of new stuff here! Let me explain:

- `-d` - "Detached" mode. Run it in the background so it keeps going
- `-p 8080:80` - "Port mapping." The container uses port 80 internally, but we want to access it via port 8080 on our computer
- `nginx` - The name of the image we're using (nginx is a popular web server)

Now let's check if it's running:

```
docker ps
```

You should see nginx in the list, running on port 8080.

Accessing Your Container

Here's the cool part. Open a web browser on your computer and type:

```
http://your-server-ip:8080
```

(Replace "your-server-ip" with your server's IP address from Chapter 3)

You should see a "Welcome to nginx" page! That's the web server running inside the container, accessible from your browser.

Stopping and Removing Containers

Let's clean up. First, stop the nginx container:

```
docker stop $(docker ps -q)
```

This stops all running containers. Now let's remove them too:

```
docker rm $(docker ps -aq)
```

Run `docker ps -a` again to confirm everything is gone.

Why This Is a Big Deal

Think about what just happened. You: 1. Installed software (nginx web server) 2. Ran it in an isolated container 3. Accessed it from your browser 4. Cleaned it up completely

And you never had to: - Install nginx manually - Configure any settings - Worry about it conflicting with other software - Figure out how to uninstall it

This is the power of Docker. Software that works, every time, zero hassle.

The Big Picture

Here's why Docker matters for self-hosting:

No more "it works on my machine" problems - If it works in the container, it'll work on your server.

Easy to try new things - Want to test a new program? Run it in a container. Don't like it? Delete it. Simple.

One command installation - Most self-hosted software provides a Docker image. One command and it's running.

Isolation - One program's problems don't affect others. If something breaks, you can delete just that container.

Easy updates - Pull a new image, delete the old container, create a new one. Done.

Do This Now

Your turn. Here's what to do right now:

1. **Make sure Docker is installed** - Run `docker --version` and celebrate if you see a version number
2. **Run the hello-world container** - Type `docker run hello-world` and watch it work
3. **Run nginx and see it in your browser** - Use `docker run -d -p 8080:80 nginx`, then open `http://your-server-ip:8080` in your browser
4. **Clean up** - Stop and remove all containers with:

   ```
   docker stop $(docker ps -q)
   docker rm $(docker ps -aq)
   ```

Take a screenshot of the nginx welcome page–you've just run your first web server!

What Comes Next

In Chapter 5, we're going to put Docker to work. You'll install AdGuard Home, which blocks ads network-wide for every device in your home. It's the perfect first service: useful immediately, impressive results, and a great way to practice your Docker skills.

But first, make sure you've played around with the commands in this chapter. Try running nginx a few more times, experiment with the `docker ps` commands, and get comfortable. Once Docker feels natural, you're ready for the fun stuff.

See you in Chapter 5!

Chapter 5: Your First Service - AdGuard Home

Finally! Let's install something useful.

In the last chapter, you installed Docker and ran a test container. That was practice. Now let's install something you'll actually use every day.

We're going to install **AdGuard Home**. Here's what it does:

It blocks ads and trackers across your entire home network.

That means every phone, tablet, computer, and smart TV in your house will get fewer ads. No browser extensions needed. No per-device setup. You set it up once, and it works for everything.

What Is AdGuard Home, Really?

Imagine you have a bouncer at the entrance of your home network. Every time a device asks, "Hey, can I load this website?" AdGuard checks a list. If the website is known for showing ads or tracking you, AdGuard says "Nope" and blocks it.

The cool part? You don't need to install anything on your devices. You just change one setting on your router (we'll get to that), and every device on your network automatically uses AdGuard.

Benefits you'll notice:

- **Fewer ads** on YouTube, websites, and apps
- **Faster loading** pages (blocked stuff doesn't load)
- **More privacy** (trackers get blocked)
- **Less data usage** on your mobile plan

Let's install it.

Step 1: Create the Folder Structure

Remember how Docker works? You need a folder to store your service's configuration. Let's create one for AdGuard.

On your server, run:

```
mkdir -p ~/docker/adguard
cd ~/docker/adguard
```

That's it. We created a folder called `adguard` inside your `docker` folder.

Step 2: Write the Setup Instructions (Docker Compose)

In the last chapter, you ran containers one at a time using `docker run`. For more complex services, we use a file called `docker-compose.yml`. This file tells Docker everything it needs to know about your service.

Create this file inside your `adguard` folder:

```
nano docker-compose.yml
```

Now paste in these lines:

```
version: "3"

services:
  adguard:
    image: adguard/adguardhome
    container_name: adguard
    restart: unless-stopped
    ports:
      - "53:53/tcp"
      - "53:53/udp"
      - "3000:3000/tcp"
    volumes:
      - ./work:/opt/adguardhome/work
      - ./conf:/opt/adguardhome/conf
```

What does all this mean? Let's break it down:

- **image**: This tells Docker what software to download (AdGuard Home)
- **container_name**: A friendly name for this container
- **restart: unless-stopped**: If your server restarts, AdGuard automatically starts again
- **ports**: These are like doorways. Port 53 is for DNS (what we're blocking ads with). Port 3000 is for the web interface where you'll manage it.
- **volumes**: These are folders that live on your server but are accessible inside the container. We need two: one for its work files and one for your settings. That way, if you delete and recreate the container, your settings stay.

When you've pasted it in, save and exit:

- Press **Ctrl + O** (the letter O, not zero)
- Press **Enter**
- Press **Ctrl + X**

Step 3: Start AdGuard

Now let's actually run it:

```
docker compose up -d
```

You should see something like:

```
[+] Running 1/1
 ? adguard  Started
```

Congratulations! AdGuard Home is running.

Step 4: Set It Up Through the Web Interface

Now comes the easy part. AdGuard has a visual interface you can use from your browser.

From a computer on the same network as your server, open your browser and type:

`http://192.168.1.X:3000`

Wait–what's the IP address?

Right. In Chapter 3, you set up your server, but let's quickly find its IP address again. On your server, run:

```
hostname -I
```

The first number shown is your server's IP address. It probably looks something like `192.168.1.100`.

So in your browser, you'd type:

`http://192.168.1.100:3000`

(Replace the numbers with whatever your server shows you.)

You should see a welcome screen. Let's finish the setup:

Step 1: Getting Started Click "Get Started."

Step 2: Web Interface Leave the settings as they are. Just note the port (3000) – that's how you'll access AdGuard later. Click "Next."

Step 3: DNS Settings Again, leave defaults. Click "Next."

Step 4: Create Admin Account This is important. Create a username and password you'll remember. This is what you'll type when you want to change your AdGuard settings. Write it down!

Click "Next" and then "Finish."

You're in! You'll see the AdGuard dashboard.

Step 5: Make Your Devices Use AdGuard

Here's the magic moment. Right now, AdGuard is installed and running. But your devices don't know to use it yet.

There are two ways to do this:

Option A: Change Your Router (Recommended)

This is the best method because it covers EVERY device automatically.

1. Open your router's settings page (usually `192.168.1.1` or `192.168.0.1`)
2. Look for "DNS" or "DHCP" settings
3. Find the place where it says "DNS Server" or "Preferred DNS"
4. Change it from whatever it is now to your server's IP address

For example, if your server is at `192.168.1.100`, you'd enter:

```
192.168.1.100
```

Save your settings.

Now every device connected to your router will use AdGuard. Phones, laptops, smart TVs, everything.

Option B: Change One Device Manually

If you can't change your router settings (some ISPs don't let you), you can change individual devices:

On Windows: 1. Go to Settings -> Network & Internet -> Wi-Fi (or Ethernet) 2. Click on your network 3. Scroll down to "DNS server" and change it to manual 4. Enter your server's IP

On Mac: 1. Go to System Settings -> Network 2. Click your Wi-Fi or Ethernet 3. Click "Details" -> "DNS" 4. Add your server's IP

On Android: 1. Go to Settings -> Network -> Wi-Fi 2. Long-press your network -> "Modify network" 3. Expand "Advanced" -> Change IP settings to "Static" 4. Enter your server's IP as DNS 1

On iPhone: 1. Go to Settings -> Wi-Fi 2. Tap the "i" next to your network 3. Tap "Configure DNS" -> "Manual" 4. Add your server's IP

Step 6: Test It Out

Now let's verify it's working.

Go back to your AdGuard dashboard in your browser (the `http://192.168.1.100:3000` page). Click on the "Dashboard" icon in the top left.

Look at the top of the page. You should see numbers going up:

- **Queries** – This counts how many times devices on your network asked for web pages
- **Blocked** – This shows how many ads or trackers got blocked

Wait a minute and refresh. The numbers should increase as your devices browse the web.

You can also test manually. Try visiting a website known for lots of ads – you should notice fewer ads than usual.

What If It Doesn't Work?

Don't panic. Here are the two most common problems:

Problem: I can't access the web interface

- Check that you typed the IP address correctly: `http://192.168.1.100:3000` (replace with YOUR server's IP)
- Make sure you added `:3000` at the end
- Is the container running? Run `docker ps` and check that "adguard" is in the list

Problem: Devices still show ads

- Make sure you changed the DNS setting on your router OR your specific device
- After changing router DNS, you may need to restart your device or toggle Wi-Fi off and on
- Check the AdGuard dashboard – are queries showing up? If not, traffic isn't going through AdGuard yet

---

What's Next?

You've now installed your first real service. This is a big deal. You:

1. ? Created a Docker Compose file
2. ? Ran a service using Docker
3. ? Configured it through a web interface
4. ? Connected devices to use it

In the next chapter, we're going to take a step back and understand how your home network actually works. This will help you troubleshoot problems and set up more services confidently.

But first – enjoy your ad-free browsing!

Do This Now

Your action step for this chapter:

1. **Install AdGuard Home** using the Docker Compose commands above
2. **Access the web interface** at `http://[your-server-ip]:3000`
3. **Create your admin account** with a password you'll remember
4. **Change one device** (your phone or laptop) to use your server's IP as its DNS server
5. **Verify it's working** – check the dashboard to see queries and blocked items

Estimated time: 15-20 minutes

This is where it starts getting fun. You've got a real service running!

Chapter 6: Understanding Your Network

Why This Chapter Matters

In the last chapter, you installed AdGuard Home and got it running on your server. You can access it from your server's web browser–but that's not very useful. You want to access it from your laptop, your phone, your smart TV–any device in your home.

To do that, you need to understand how devices find each other on your network. This is the missing piece that confuses most beginners, and it's actually simpler than you think.

Think of your home network like a small town. Your router is the post office. Every house has a unique address. And your server is one of the houses–waiting to receive mail (requests) and send back responses.

Let's demystify this.

Your Network Is Like a Neighborhood

Imagine a small neighborhood with houses on it. Each house has a unique house number:

- House #1: The family laptop
- House #2: Your phone
- House #3: Your server (where AdGuard lives)
- House #4: Your smart TV

When your phone wants to ask your server "Hey, show me the AdGuard dashboard," it needs to know where the server lives. That's where IP addresses come in.

What Is an IP Address?

An **IP address** is just a unique number that identifies a device on your network. It's like a house address, but for computers.

In most home networks, IP addresses look something like this: `192.168.1.100`

Break it down: - `192.168.1` – this part is your network (every device in your home shares this) - `.100` – this last number is the unique identifier for a specific device

So your server might be `192.168.1.100`, your phone might be `192.168.1.105`, and so on.

These are called **local IP addresses** because they only work inside your home network. That's why you can't type your server's IP address into your phone

when you're at a coffee shop–it won't work. Your phone isn't on your home network anymore.

Finding Your Server's Address

Now let's find your server's IP address. This is something you'll need often, so learn it now.

On Your Server (Ubuntu)

Run this command on your server:

```
ip a
```

Look for the section called `eth0` or `ens` followed by numbers. You'll see something like:

```
inet 192.168.1.100/24
```

That `192.168.1.100` is your server's local address. Write it down–you'll use it constantly.

The Easy Way: Check Your Router

If that command feels intimidating, here's an alternative:

1. Log into your router (usually by typing `192.168.1.1` in a browser)
2. Look for a section called "Connected Devices" or "DHCP Clients"
3. You'll see a list of all devices on your network, including your server

Your router's manual or the sticker on the bottom will tell you the login credentials.

Ports: The Doors of Your House

Now you know the address. But there's one more piece: the **port**.

Think of your server as an apartment building with many doors. Each door (port) leads to a different service:

- Port 80: The front door (regular websites)
- Port 443: The secure front door (websites with the lock icon)
- Port 3000: AdGuard's specific door
- Port 53: The DNS door (what AdGuard uses)

When you type `192.168.1.100:3000` into your browser, you're saying: "Go to house number 192.168.1.100, and use door number 3000."

That's it.

Common Ports You'll Encounter

Service	Port	What It's For
HTTP (regular website)	80	Viewing websites
HTTPS (secure website)	443	Viewing secure websites
AdGuard Home	3000	Its web interface
SSH	22	Logging into your server remotely
Plex/Jellyfin	32400 / 8096	Media servers

You don't need to memorize these. But when you install new services, you'll often need to know which port they use–so always check the installation instructions.

How Devices Talk to Each Other

Here's the flow when your phone accesses AdGuard:

1. Your phone says: "I want to talk to 192.168.1.100, port 3000"
2. Your router (the post office) forwards that request to your server
3. Your server's AdGuard service receives it
4. AdGuard responds with the data your phone needs
5. The data travels back through the router to your phone

This happens in milliseconds. You won't notice any delay.

Local vs. Public: The Key Distinction

This is where most people get confused.

Local access (from inside your home): - You type `192.168.1.100:3000` on your laptop - It works perfectly - Your request never leaves your house

Public access (from outside your home): - You try to type your home IP address from a coffee shop - It doesn't work - Your request has no way to find your home network

We'll cover how to access your services from outside your home in Chapter 8. For now, focus on getting comfortable accessing services from devices *inside* your network.

Accessing AdGuard from Another Device

This is the moment you've been building toward. Let's actually do it.

Do This Now: Access AdGuard from Your Laptop

Step 1: Make sure your laptop is connected to the same WiFi network as your server.

Step 2: Open a web browser on your laptop (Chrome, Firefox, Safari–whatever you prefer).

Step 3: In the address bar, type:

`http://192.168.1.100:3000`

(Replace `192.168.1.100` with whatever your server's IP address actually is)

Step 4: Press Enter.

You should see the AdGuard Home login screen! If you set up a password in Chapter 5, log in. If not, you're directly in.

Step 5: Try accessing it from your phone too.

- Connect your phone to the same WiFi
- Open your phone's browser
- Type the same address: `http://192.168.1.100:3000`

You now have network-wide ad blocking working from every device in your house. That's powerful!

Troubleshooting

If it doesn't work:

1. **Double-check the IP address** – run `ip a` on your server again to confirm
2. **Check the port** – make sure you typed `:3000` (the colon matters!)
3. **Are you on the same network?** – make sure laptop and server share the same WiFi
4. **Is Docker running?** – run `docker ps` to confirm AdGuard is actually running

Don't panic if it doesn't work immediately. Networking can be finicky. Check each item systematically, and you'll find the issue.

What You've Learned

Let's recap what this chapter covered:

- **IP addresses** are like house numbers for devices on your network
- **Local IP addresses** only work inside your home
- **Ports** are like doors–each service uses a different one
- Your router acts as the post office, directing traffic to the right place
- Accessing services by IP:port lets you reach any device on your network

You now have the foundation to understand how your entire home network works. This knowledge applies to every service you'll install going forward.

What's Next

In the next chapter, we'll build on this knowledge. You'll install two more services: Jellyfin (for your personal Netflix) and Vaultwarden (for a password manager that you control).

But first, take a moment to appreciate what you just accomplished. You installed a network-wide ad blocker that works on every device in your house–from your laptop to your smart TV. That's something most people never figure out.

You just did.

Quick Reference

Finding Your Server's IP

```
ip a
```

Checking Running Services

```
docker ps
```

Accessing AdGuard

```
http://[YOUR_SERVER_IP]:3000
```

Checking Your Public IP (for curiosity)

```
curl ifconfig.me
```

"Do This Now" Summary

? **Done:** Found your server's IP address using `ip a`
? **Done:** Accessed AdGuard from your laptop browser
? **Done:** Accessed AdGuard from your phone

If you completed all three, congratulations–you understand your network!

Chapter 7: Adding More Services - Media & Passwords

Your server can do more than block ads.

In the last chapter, you installed AdGuard Home – a service that runs in the background, protecting your whole network. Now let's install two services that you'll actually interact with every day.

We're going to install:

1. **Jellyfin** – a media server that lets you stream your own videos, music, and photos to any device
2. **Vaultwarden** – a password manager that keeps your passwords secure and accessible only to you

Both are incredibly useful, and both demonstrate something important: how Docker stores your data so it doesn't disappear when you restart things.

Let's start with Jellyfin.

Part 1: Jellyfin – Your Personal Netflix

What Does a Media Server Do?

Imagine you have a box of DVDs and Blu-rays in your closet. Every time you want to watch something, you have to: 1. Find the right disc 2. Put it in your player 3. Hope it doesn't scratch

Now imagine all those movies are in one place, and you can watch any of them on any device – your phone, laptop, smart TV, tablet – without getting up. That's what a media server does.

Jellyfin scans a folder on your server, finds all your videos and music, and presents them in a beautiful interface. Think of it as your own private Netflix.

Why self-host this? - Stream your movie collection anywhere - No subscription fees (unlike Netflix, Hulu, etc.) - Keep full control of your content - Watch on multiple devices simultaneously

Step 1: Create the Folder and Compose File

Just like with AdGuard, we need a folder for Jellyfin:

```
mkdir -p ~/docker/jellyfin
cd ~/docker/jellyfin
```

Now let's create the Docker Compose file. This time, it will be a bit longer because we need to set up storage for your media files:

```
nano docker-compose.yml
```

Paste in these lines:

```
version: "3"

services:
  jellyfin:
    image: jellyfin/jellyfin
    container_name: jellyfin
    restart: unless-stopped
    ports:
      - "8096:8096"
      - "8920:8920"
    volumes:
      - ./config:/config
      - ./cache:/cache
      - /home/yourusername/media:/media
    environment:
      - TZ=America/New_York
```

Wait – this looks different from AdGuard. Let's break it down:

image: `jellyfin/jellyfin` – this is the software we're downloading

ports: - Port 8096 is for the web interface (where you'll watch videos) - Port 8920 is for secure access (HTTPS), which we'll set up later

volumes (this is the important part): - `./config:/config` – stores your Jellyfin settings, user accounts, and library information - `./cache:/cache` – temporary files that help Jellyfin run faster - `/home/yourusername/media:/media` – THIS is where your videos go. The part before the colon (`/home/yourusername/media`) is a folder ON YOUR SERVER. The part after the colon (`/media`) is where Jellyfin looks inside its container.

Important: Change `/home/yourusername/media` to your actual home folder path. If your username is "bob", it would be `/home/bob/media`.

environment: We're setting the timezone so Jellyfin shows the right time

Step 2: Create Your Media Folder

Now create the folder where you'll put your videos:

```
mkdir -p /home/yourusername/media
```

(Replace "yourusername" with your actual username on the server.)

Step 3: Start Jellyfin

Let's launch it:

```
docker compose up -d
```

You should see:

```
[+] Running 1/1
 ? jellyfin  Started
```

Step 4: Set Up Jellyfin Through the Web

Now for the fun part. Open your browser and go to:

```
http://192.168.1.100:8096
```

(Replace `192.168.1.100` with your server's IP address.)

You'll see a welcome screen. Let's set it up:

Step 1: Choose Your Language Select your language and click "Next"

Step 2: Create Admin Account Pick a username and password. This is for managing Jellyfin (adding users, changing settings). Click "Next"

Step 3: Add Media Library Click "Add Media Library"

- **Content Type:** Movies
- **Display Name:** Movies (or whatever you want)
- **Folder:** Click "Browse" and select the `/media` folder we created earlier

Click "OK" and then "Next"

Step 4: Preferred Metadata Language Pick your language and click "Next"

Step 5: Enable Remote Access You can leave this as-is for now. We'll cover remote access in Chapter 8. Click "Next"

Step 6: Complete! Click "Finish"

You're in! You'll see the Jellyfin dashboard.

Step 5: Add a Video File

Now let's add an actual video to watch.

First, copy a video file (MP4, MKV, or AVI) to your media folder on the server:

```
cp /path/to/your/video.mp4 /home/yourusername/media/
```

(Replace `/path/to/your/video.mp4` with the actual location of a video file on your server.)

Now go back to your Jellyfin browser tab. Click on "Movies" in the sidebar. You should see your video appear!

Click on it and press Play. You're streaming your own video through your own server.

Part 2: Vaultwarden – Your Personal Password Manager

What Is Vaultwarden?

Every website you use – banking, email, social media – should have a different password. That's common knowledge. But remembering 50+ unique passwords? That's impossible without help.

Most people use password managers like 1Password or LastPass. These are services that: 1. Store all your passwords in an encrypted "vault" 2. Generate strong, random passwords for you 3. Automatically fill in passwords when you visit websites

The problem? Those services store your passwords on THEIR servers. You have to trust them completely.

Vaultwarden is the self-hosted alternative. It does everything the big name managers do, but the data lives on YOUR server. No subscription fees. Complete privacy.

Why self-host this? - No monthly fee (free forever) - Your passwords never leave your network - Syncs across all your devices - Open source (anyone can verify it's secure)

Step 1: Create the Folder

```
mkdir -p ~/docker/vaultwarden
cd ~/docker/vaultwarden
```

Step 2: Create the Docker Compose File

```
nano docker-compose.yml
```

Paste in:

```
version: "3"

services:
  vaultwarden:
    image: vaultwarden/server
    container_name: vaultwarden
    restart: unless-stopped
    ports:
      - "8080:80"
    volumes:
      - ./data:/data
    environment:
      - SIGNUP_ALLOWED=true
```

Let's explain what's new:

image: `vaultwarden/server` – this is an open-source implementation compatible with Bitwarden (a popular password manager)

ports: Port 8080 for the web interface

volumes: `./data:/data` – this stores your password vault. This is the most critical folder because it contains all your passwords, encrypted.

environment: `SIGNUP_ALLOWED=true` – this lets you create an account. Once you've set up your account, you might want to change this to `false` to prevent other people from signing up.

Step 3: Start Vaultwarden

```
docker compose up -d
```

Step 4: Create Your Account

Open your browser and go to:

`http://192.168.1.100:8080`

(Again, replace with your server's IP.)

You should see the Vaultwarden login page. Click "Create Account"

Fill in: - **Email:** Your email address - **Master Password:** This is THE most important password. It encrypts your entire vault. If you forget this, you CANNOT recover your passwords. Write it down somewhere safe! - **Master Password Hint:** A hint to help you remember

Click "Submit"

You're in! This is your password vault.

Step 5: Install Browser Extension (Optional but Recommended)

To get the full benefit, install the browser extension:

1. Go to the Vaultwarden website in your browser
2. Look for "Browser Extensions"
3. Install the extension for Chrome/Firefox/Edge
4. When it asks for the server URL, enter: `http://192.168.1.100:8080`

Now you can: - Save passwords when you create accounts - Auto-fill passwords when you log in to websites - Check if your passwords have been exposed in data breaches

---

Understanding Volumes – Why Your Data Doesn't Disappear

You might have noticed something: both Jellyfin and Vaultwarden have folders that start with `./` in their volumes section.

Let's talk about why this matters.

The Problem

When you run a container, it exists only as long as it's running. If you delete the container, everything inside it disappears. All your settings, all your data – gone.

That's obviously a problem. You don't want to lose your password vault every time you update the software.

The Solution: Volumes

A **volume** is like a bridge between your server's hard drive and the inside of the container.

Think of it like this:

- The container is a house
- Your server's hard drive is a storage unit down the street
- A volume is a teleport hallway connecting them

Data that goes through this hallway gets saved on your hard drive. Even if the house (container) gets destroyed and rebuilt, the stuff in the storage unit (your hard drive) is still there.

That's why we wrote:

```
volumes:
  - ./data:/data
```

The `./data` part means "a folder called 'data' in the same folder as this docker-compose.yml file". The `/data` part is where the container expects to find its data.

So when Vaultwarden saves a password, it goes into `/data` inside the container, which actually saves it to the `data` folder on your server. Pretty cool, right?

This Is a Big Deal

Understanding volumes is one of the most important things about self-hosting. It means:

- **You can update** your services without losing data
- **You can move** your services to a different server and take your data with you

- **You can back up** your data by copying these folders

We'll talk more about backups in Chapter 10.

What If It Doesn't Work?

Jellyfin problems:

- *Video won't play:* Make sure the video file is in your `/media` folder and is a supported format (MP4, MKV, AVI work best)
- *Can't see videos:* Click the refresh icon in Jellyfin to rescan your media library
- *Permission denied:* Make sure the media folder is readable. Run `chmod -R 755 /home/yourusername/media`

Vaultwarden problems:

- *Can't create account:* Check that you set `SIGNUP_ALLOWED=true`
- *Extension can't connect:* Make sure you entered the correct server URL (including the port number)
- *Forgot master password:* Unfortunately, there's no recovery. You'll need to delete the data folder and start fresh

---

What's Next?

You've now installed three services:

1. **AdGuard** – blocks ads network-wide
2. **Jellyfin** – streams your media
3. **Vaultwarden** – manages your passwords

That's a solid foundation! But there's one thing we haven't figured out yet: how do you access these services when you're NOT at home?

In the next chapter, we're going to understand your network better. You'll learn about IP addresses, ports, and how devices talk to each other. This knowledge is essential for accessing your services remotely.

But first – go watch that video you added to Jellyfin. You earned it!

Do This Now

Your action steps for this chapter:

1. **Install Jellyfin** using the Docker Compose commands above

2. **Create your media folder** (`mkdir -p /home/yourusername/media`)
3. **Copy one video file** to that folder
4. **Set up Jellyfin** through the web interface at `http://[your-server-ip]:8096`
5. **Watch the video** in your browser to confirm it works
6. **Install Vaultwarden** using the Docker Compose commands
7. **Create your Vaultwarden account** at `http://[your-server-ip]:8080`
8. **Add one password** to test it works

Estimated time: 30-40 minutes

Take your time with this one. Getting media streaming working for the first time is incredibly satisfying!# Chapter 8: Accessing Your Services From Outside

Estimated reading time: 15 minutes

The Problem You're Facing

By now, you've got services running on your server at home. AdGuard is blocking ads. Jellyfin is streaming your videos. Vaultwarden is keeping your passwords safe. Life is good.

But here's the catch: all of this only works when you're on your home WiFi.

What happens when you're at work and need a password from Vaultwarden? Or you're traveling and want to watch a movie from your Jellyfin library? Or you're at a coffee shop and want to check on your home network?

This chapter answers the most common question beginners ask: **"How do I access my stuff when I'm not at home?"**

Why Can't You Just Open a Port?

You might think: "Why not just open my server to the internet and type in my home IP address?"

Technically, you could do that. But there are three big problems:

1. **Your home IP address changes.** Internet providers periodically change your public IP address. Tomorrow, your "front door" might be at a completely different number. You can't memorize a changing address.

2. **It's not secure.** Opening a port directly to the internet is like leaving your front door unlocked. Anyone can try to get in. Hackers constantly scan the internet for open ports and try to guess passwords.

3. **No encryption.** When you access your server from outside, your data travels through the public internet. Without protection, anyone on the

same WiFi as you (or monitoring network traffic) can see your passwords and sensitive information.

So we need a better solution. Let's look at your options.

Your Three Options (In Plain English)

Option 1: VPN - The Secure Tunnel

Think of a VPN (Virtual Private Network) like a secret tunnel that connects your phone directly to your home network. When you use a VPN, your phone "pretends" to be on your home WiFi, even when you're thousands of miles away.

How it works: You install VPN software on both your server and your phone. When you connect, your phone creates an encrypted tunnel to your home. All your traffic goes through this tunnel, like a private highway that nobody else can use.

Pros: - Very secure (encryption is built-in) - Works with all services (no special configuration per service) - You don't need to buy a domain name

Cons: - Slight learning curve to set up - Need to remember to turn on the VPN before accessing your services

Option 2: Reverse Proxy - The Reception Desk

A reverse proxy is like having a receptionist at your home. When someone from the outside world wants to visit, they don't go directly to your server. Instead, they go to your "receptionist" (the proxy), who then directs them to the right service.

This is how websites work. But for home services, you'd typically buy a domain name (like myhome.dynv6.net) and point it to your home.

How it works: When you type yourdomain.com/vaultwarden, the proxy sees the "/vaultwarden" part and sends you to your password manager. When you type yourdomain.com/jellyfin, it sends you to your media server.

Pros: - Easy to remember URLs (yourdomain.com instead of an IP address) - Professional feel - Works great when you need to share access with others

Cons: - Requires buying a domain name (costs money) - More complex to set up - Need to handle SSL certificates for security

Option 3: Cloudflare Tunnel - The Smart Shortcut

Cloudflare Tunnel is a newer approach that doesn't require opening any ports on your home network. Instead, your server connects OUT to Cloudflare's servers,

creating a secure path. When you want to access your services, you go through Cloudflare.

How it works: You install a small program on your server that creates a permanent connection to Cloudflare. Cloudflare then handles the outside world, routing traffic to your home through that connection. You never open any ports.

Pros: - No port forwarding needed (more secure by default) - Built-in security features from Cloudflare - Works without a static IP

Cons: - Need to use Cloudflare's ecosystem - Slight dependency on an external service - Can be confusing to set up for beginners

---

Which One Should You Start With?

For most beginners, we recommend **Tailscale** (a VPN built on WireGuard). Here's why:

1. **Easiest to set up.** Tailscale was designed to be simple. Install an app on your phone, run a command on your server, and you're done. No router configuration, no domain names, no port forwarding.
2. **Works automatically.** Once set up, you don't need to think about it. Your phone is always "on" your home network when you turn on the VPN.
3. **Free for personal use.** Tailscale offers a free tier that's more than enough for personal self-hosting.
4. **Encryption built-in.** All your traffic is encrypted automatically. Safe on public WiFi.

Think of it this way: - **VPN (Tailscale)** = Direct secure tunnel to your home - **Reverse Proxy** = Professional setup with custom domain (good for sharing with others) - **Cloudflare Tunnel** = No-ports-needed approach (good for complex setups)

Start with Tailscale. Once you're comfortable, you can explore the others.

What You'll Need

Before we start, make sure you have: - Your server running (from Chapter 3) - A phone, tablet, or laptop you want to use to access your services remotely - An email address (for Tailscale account) - About 15 minutes

---

Installing Tailscale on Your Server

Let's start by installing Tailscale on your server. This is the "home base" end of your secure tunnel.

Step 1: Install Tailscale

Copy and paste this command into your server's terminal:

```
curl -fsSL https://tailscale.com/install.sh | sh
```

This downloads and installs the Tailscale software. You'll see some text scroll by as it installs.

Step 2: Connect to Your Network

Once installed, tell Tailscale to connect to your account:

```
sudo tailscale up
```

This will open a web browser link (or give you a URL to visit). Click it, sign in with your email, and approve your device.

That's it! Your server is now part of your private Tailscale network.

Step 3: Note Your Tailscale IP

Tailscale will show you an IP address that starts with `100.`. This is your server's address on your private network. Write it down–you'll need it.

For example: `100.64.123.456`

Installing Tailscale on Your Phone

Now let's get your phone connected.

On iPhone (App Store) or Android (Play Store):

1. Search for "Tailscale" and install the app
2. Open the app and sign in with the same account you used on your server
3. Tap "Start"

Your phone is now connected!

On Computer (Mac/Windows/Linux):

1. Go to https://tailscale.com/download
2. Download and install the app for your operating system
3. Sign in with the same account
4. Turn on the connection

Testing Your Connection

Now let's verify that everything works.

From your phone:

1. Make sure you're NOT on your home WiFi (turn off WiFi and use mobile data, or go to a friend's house)
2. Open your browser and type the Tailscale IP you wrote down earlier (the `100.` address)
3. Add the port number for a service you know is running

For example: `http://100.64.123.456:8080`

- Port 8080 = AdGuard (if you set it up in Chapter 5)
- Port 8096 = Jellyfin (if you set it up in Chapter 7)

You should see your service! ?

Try this:

- Open Vaultwarden (port 8081) and log in
- Open Jellyfin (port 8096) and play a video
- Open AdGuard (port 8080) and check your statistics

You're now accessing your home services from anywhere in the world, through an encrypted tunnel. The data between your phone and your server is completely private.

How to Find Your Service Ports

In the previous chapters, we mentioned ports but didn't explain them in detail. Here's a quick reference:

When you set up a service with Docker (like in Chapter 5 and 7), the docker-compose.yml file specifies which port to use. You can always check:

- In your docker-compose.yml file, look for `ports:` - the number after the colon is the port
- Or run `docker ps` on your server to see all running containers and their ports

Common ports you'll use: - 8080 = AdGuard Home - 8081 = Vaultwarden - 8096 = Jellyfin

---

Understanding What Just Happened

Let me explain what you just built:

```
[Your Phone] ---> [Internet] ---> [Tailscale Tunnel] ---> [Your Server]
                  (encrypted)                             (decrypts here)
```

When you access your server through Tailscale: 1. Your phone encrypts your request 2. It goes through the public internet to Tailscale's servers 3. Tailscale routes it to your home server through the tunnel you created 4. Your server decrypts it, processes your request, and sends back the answer 5. The answer comes back through the same encrypted tunnel

Anyone watching the internet traffic sees only gibberish. They can't see your passwords, your videos, or which services you're accessing.

What Tailscale Doesn't Work?

Sometimes If, home internet routers have settings that interfere. If your connection doesn't work:

1. **Check that Tailscale is running on both devices** - Look for the green indicator in the app
2. **Check your firewall** - Some routers have built-in firewalls that might block Tailscale. Try accessing from a different network (like your phone's mobile data)
3. **Check the IP address** - Make sure you're using the Tailscale IP (the 100.x.x.x address), not your home's public IP
4. **Restart Tailscale** - Run `sudo tailscale down` then `sudo tailscale up` on your server

Most connection issues resolve themselves within a few minutes of setup.

When Would You Need a Domain Name?

You might be wondering: "Do I ever need to buy a domain name?"

Not for Tailscale. The 100.x.x.x address works forever.

But you might want a domain name if: - You want to share access with friends and family (easier to remember than an IP) - You want professional-looking URLs - You want to set up multiple services with nice addresses (like vault.myhouse.com instead of 100.x.x.x:8081)

We'll cover domain names and reverse proxies in Chapter 9, once you've got the basics down.

What's Next?

You now have secure, encrypted access to all your home services from anywhere in the world. This is a huge milestone!

In the next chapter, we'll talk about security. Now that you can access your services remotely, we need to make sure nobody else can. We'll cover: - Setting up a reverse proxy (Nginx Proxy Manager) - Adding automatic HTTPS (SSL certificates) - Adding password protection to your services

But first, let's make sure you can actually access your services. That's what we'll practice right now.

Do This Now

Your mission, should you choose to accept it:

1. **Install Tailscale on your server** (run the install command from this chapter)
2. **Connect it to your account** (run `sudo tailscale up` and follow the link)
3. **Install Tailscale on your phone** (or another device)
4. **Test remote access** - Turn off your home WiFi, use mobile data, and access one of your services

Take a screenshot of yourself accessing Jellyfin or Vaultwarden from outside your home. This is a real achievement!

Expected time: 15-20 minutes
Difficulty: Easy
You'll feel like a hacker. ?

Quick Reference

Useful Tailscale Commands (for your server)

```
# Check status
tailscale status

# Turn off Tailscale
sudo tailscale down
```

```
# Turn on Tailscale
sudo tailscale up

# See your Tailscale IP
tailscale ip -4
```

Troubleshooting Checklist

Problem	Solution
Can't connect	Make sure Tailscale is running on both devices
Wrong page loads	Check you're using the right port number
Connection slow	Try a different network (mobile vs WiFi)
Services not running	Check with `docker ps` that your services are actually running

---

Recap

In this chapter, you learned:

- Why opening ports directly to the internet is a bad idea
- The three main ways to access your services remotely: VPN (Tailscale), reverse proxy, and Cloudflare Tunnel
- Why Tailscale is the easiest starting point for beginners
- How to install and configure Tailscale on both your server and phone
- How to test your remote access connection

You now have the power to access your entire home server from anywhere. This opens up a world of possibilities–checking your security cameras, grabbing a password, streaming your media, all from your phone.

In the next chapter, we'll make sure nobody else can access your services without permission. Security time! ?

Chapter 9: Securing Your Setup

Estimated reading time: 15 minutes

In the last chapter, you learned how to access your services from anywhere using Tailscale. That's incredibly useful–but here's an important truth: anytime you open a door to the outside world, you need a lock on it.

Think of it like your home. You want guests to visit, so you have a front door. But you don't leave it wide open, do you? You have a lock, maybe a doorbell camera, and you probably ask who someone is before letting them in.

The same thinking applies to your self-hosted services. In this chapter, you'll learn how to:

- Set up a secure gateway that handles all incoming traffic (Nginx Proxy Manager)
- Encrypt the connection so nobody can spy on your data (SSL/HTTPS)
- Add password protection as a first line of defense
- Set up a firewall to block unwanted visitors

Let's get your setup locked down.

What Does "Securing" Actually Mean?

Before we dive in, let's clarify what we're actually protecting against.

When you expose a service to the internet–even with a VPN like Tailscale–you're creating a potential entry point. Here's what can go wrong if you don't secure things:

1. Unencrypted connections = Anyone can read your data

When you access a service over plain HTTP (without the "S"), your password, messages, and data travel as plain text. Imagine sending a postcard instead of a sealed letter–anyone who handles it can read it.

2. No password protection = Anyone can walk right in

Some services come with no default password at all. If someone guesses the web address, they can access everything.

3. Open ports = An unlocked window

Every service that listens on the internet has a "port." Leaving ports open without protection is like leaving windows open with no screens.

The good news? You don't need to be a security expert. We'll use tools that handle most of this automatically.

Meet Your Security Guard: Nginx Proxy Manager

In Chapter 8, you learned about reverse proxies–a piece of software that sits in front of your services and directs traffic. Nginx Proxy Manager (often abbreviated as NPM) is that same idea, but with superpowers.

NPM does three important things:

1. **Routes traffic** - It receives incoming requests and sends them to the right service (like a receptionist directing visitors)
2. **Adds encryption automatically** - It fetches and manages SSL certificates (we'll explain what those are in a moment) without you needing to do anything technical
3. **Adds password protection with one click** - You can require a username and password before anyone even reaches your service

Think of Nginx Proxy Manager as a combination of a receptionist, an encrypted tunnel builder, and a bouncer. Pretty powerful, right?

Installing Nginx Proxy Manager

NPM runs as a Docker container, just like the services you've already installed. Here's the docker-compose.yml:

```yaml
version: '3.8'

services:
  npm:
    image: 'jc21/nginx-proxy-manager:latest'
    container_name: nginx-proxy-manager
    restart: unless-stopped
    ports:
      - '80:80'       # HTTP traffic
      - '443:443'   # Encrypted HTTPS traffic
      - '81:81'      # NPM's own admin interface
    volumes:
      - ./data:/data
      - ./letsencrypt:/etc/letsencrypt
```

Save this as `docker-compose.yml` in a new folder called `nginx-proxy-manager`, then run:

```
docker-compose up -d
```

Give it a minute to start up, then open your browser and go to:

```
http://YOUR_SERVER_IP:81
```

You'll see the NPM login screen. The default credentials are:

- **Email:** admin@example.com
- **Password:** changeme

Change these immediately after logging in!

What Is SSL/HTTPS? (The Simple Version)

You know how some website addresses start with "http://" and others with "https://"? That extra "s" stands for "secure." But what does it actually mean?

Without HTTPS (http://): When your phone sends your password to your server, it's like whispering it across a crowded room. Anyone close enough can hear it.

With HTTPS (https://): It's like speaking in a secret language that only you and your server understand. Even if someone overhears, they can't understand anything.

The "secret language" is enabled by something called an SSL certificate. It's not as complicated as it sounds–it's just a small file that proves your server is who it claims to be, and it encrypts everything in transit.

Here's the beautiful part: Nginx Proxy Manager can get these certificates for free, automatically, from a nonprofit called Let's Encrypt. You don't need to pay anything or understand the technical details.

Enabling HTTPS for a Service

Once NPM is running, here's how to add HTTPS to any service:

Step 1: In NPM, click "Proxy Hosts" -> "Create Proxy Host"

Step 2: Fill in the details:

- **Domain Name:** The web address people will use (e.g., jellyfin.yourhome.com)
- **Forward IP:** Your server's local IP address (e.g., 192.168.1.100)
- **Forward Port:** The port your service runs on (e.g., 8096 for Jellyfin)

Step 3: Scroll down and check these options:

- **Force SSL:** This automatically redirects anyone trying to use plain HTTP to the secure version
- **Enable SSL/HTTPS:** Click this and select "Request a new SSL Certificate"

Step 4: Enter your email address (Let's Encrypt uses this to remind you when certificates are due for renewal)

Step 5: Click Save

That's it. NPM will contact Let's Encrypt, get the certificate, and configure everything. Within seconds, your service is accessible via HTTPS.

You'll see a little lock icon in your browser's address bar. That lock means the connection is secure.

Adding Basic Password Protection

Even with HTTPS, you might want an extra layer of security. Maybe you haven't set up strong passwords on all your services yet, or you want to make absolutely sure only you can access them.

NPM lets you add "Basic Authentication" with just a few clicks. This means anyone trying to access a service must enter a username and password that you define–not the service's own login.

Setting It Up

Step 1: In NPM, go to the proxy host you created earlier and click "Edit"

Step 2: Scroll to the "Basic Authentication" section

Step 3: Check "Enable Basic Authentication"

Step 4: Click "Add" to create a user. Enter: - **Username:** (pick something like "admin" or "family") - **Password:** (pick a strong password)

Step 5: Click Save

Now when anyone tries to access that service–even from inside your home network–they'll see a popup asking for the username and password you just created.

Important: This is an extra layer, not a replacement for good passwords on your actual services. Think of it like a security gate outside your house–you still want locks on your doors inside.

Your Digital Bouncer: The Firewall

Now let's talk about the firewall. If NPM is the receptionist and the bouncer, the firewall is the walls around your property. It decides which doors exist and who's allowed to knock on them.

A firewall monitors all incoming and outgoing network traffic and blocks anything that doesn't meet your rules. For a home server, you typically want to:

- **Allow** traffic you've specifically requested (like your own web browsing)

- **Block** incoming traffic unless it's for a service you want publicly accessible

The Simple Firewall Setup

For most home self-hosters, you don't need complex firewall rules. Here's the practical approach:

1. Only open ports you need

Remember when we talked about ports in Chapter 6? Each service uses a specific port. Instead of opening everything, only open:

- Port 80 (HTTP)
- Port 443 (HTTPS)

These are the standard web ports. NPM will handle routing traffic to your services from these ports–you don't need to open individual ports for each service.

2. Use UFW (Uncomplicated Firewall)

Ubuntu comes with a firewall tool called UFW (Uncomplicated Firewall–computer people love acronyms). It's already installed; you just need to turn it on.

Run these commands:

```
# Check status
sudo ufw status

# Allow SSH (so you don't lock yourself out!)
sudo ufw allow ssh

# Allow HTTP and HTTPS
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

# Turn on the firewall
sudo ufw enable
```

Warning: Never lock yourself out of SSH! Always allow SSH (port 22) before enabling the firewall.

3. Check who tried to visit

You can see blocked attempts with:

```
sudo ufw status numbered
```

You might be surprised how many automated bots are constantly scanning the internet, looking for weak spots. The firewall stops them at the door.

Security Layers: How It All Fits Together

Let's visualize your security setup now:

```
Internet
    ?
[Firewall - Blocks unwanted ports]
    ?
[Nginx Proxy Manager - Routes to correct service]
    ?
[SSL Certificate - Encrypts the connection]
    ?
[Basic Auth - Password gate]
    ?
[Your Service - The actual app]
```

Each layer does a specific job. Even if one layer fails, others are there as backup. This is called "defense in depth"–a fancy term for "lots of locks are better than one."

Here's the order of what a random stranger on the internet would encounter:

1. **Firewall:** "Sorry, the door is closed."
2. **Without proxy:** They might find an open port but won't know which service it leads to
3. **With NPM:** They reach the right service, but no SSL means the connection is visible
4. **With SSL:** The data is encrypted, but they'd still need to guess your password
5. **With Basic Auth:** They need your username and password just to get through the door

That's five layers of defense. Not bad for a beginner setup!

Common Mistakes to Avoid

Before we move to the exercise, let's cover some pitfalls:

Mistake #1: Opening too many ports

More open ports = more potential entry points. Only open what you need, and route everything through NPM.

Mistake #2: Using weak passwords

If your basic auth password is "password123," it's not really protection. Use a password manager (you installed Vaultwarden in Chapter 7, right?) to generate strong, unique passwords.

Mistake #3: Ignoring certificate renewals

Let's Encrypt certificates expire after 90 days. The good news: NPM automatically renews them. Just don't stop the container for too long!

Mistake #4: Turning off the firewall "because it's annoying"

Yes, sometimes the firewall blocks things you want. But it's protecting you. Learn which rules to add instead of turning it off.

Do This Now

Time to put this chapter into action. Here's your exercise:

Set Up Nginx Proxy Manager with SSL

What you'll accomplish: Install NPM, add one of your existing services (like Jellyfin or Vaultwarden), and access it via HTTPS with basic authentication.

Steps:

1. **Create the NPM folder and docker-compose.yml** (use the code from earlier in this chapter)
2. **Start the container:**

   ```
   cd nginx-proxy-manager
   docker-compose up -d
   ```

3. **Access NPM admin interface** at `http://YOUR_SERVER_IP:81` and change the default password
4. **Create a proxy host** for one of your existing services:
   - Use a simple subdomain like `jellyfin.yourhome.local` (you don't need a real domain for local use)
   - Forward to your server's IP and the service's port
   - Enable SSL and select "Request a new SSL Certificate"
   - Add basic authentication with a username and strong password
5. **Test it:** Access the service through the new URL. You should see:
   - A lock icon in your browser (SSL working)
   - A popup asking for username/password (basic auth working)

Congratulations! You've just secured your first service. This same process works for any other services you want to expose.

What's Next?

In Chapter 10, we'll cover the unsexy but essential topic of maintenance. How do you update services without breaking things? How do you back up your data? What happens if your server crashes?

"Set and forget" is a myth–but with the right maintenance habits, you can keep your setup effort.

See you there! running smoothly with minimal# Chapter 10: Keeping It Running

Welcome to the chapter nobody talks about.

By now, you've built something great. You have AdGuard blocking ads, Jellyfin streaming your videos, Vaultwarden keeping your passwords safe. Everything works. Life is good.

But here's the truth nobody tells you when you start self-hosting: **your server needs love.** Things break. Updates come out. Sometimes your server just needs a restart. And if you're not paying attention, you might not even know something is wrong until it's been broken for days.

The good news? Setting up proper maintenance doesn't take long. Once it's in place, your server essentially runs itself. You'll sleep better at night knowing that if something goes wrong, you'll find out quickly.

Let's cover the three pillars of keeping your server healthy: updating, backups, and monitoring.

Updating Your Services (Without Breaking Everything)

Think about the apps on your phone. Every few weeks, they ask you to update–sometimes for new features, often for security fixes. Your self-hosted services work the same way.

Developers constantly release new versions to:

- Fix security holes (important!)
- Add new features
- Repair bugs you might not even know existed

The challenge is that updates can sometimes change how things work. That's why we update carefully.

The Simple Way: Watchtower

Remember how Docker lets you run services in standardized packages? There's a tool called **Watchtower** that automatically checks for updates to your containers and notifies you–or even updates them automatically.

Here's how simple it is to add Watchtower to your setup:

```yaml
services:
  watchtower:
    image: containrrr/watchtower
    container_name: watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - WATCHTOWER_SCHEDULE=0 0 4 * * *
      - WATCHTOWER_NOTIFICATIONS=shoutrrr
      - WATCHTOWER_NOTIFICATION_URL=shoutrrr://...
```

That schedule `0 0 4 * * *` means "check for updates every day at 4 AM." You don't need to remember to check–Watchtower does it for you.

The Even Simpler Way: Just Check Manually

If that sounds like too much automation (some people love being hands-on), you can manually check for updates. Here's the command:

```
docker-compose pull
docker-compose up -d
```

The `pull` command downloads the newest version of each image. The `up -d` command restarts your services with the new version. The `-d` means "run in detached mode," which is just a fancy way of saying "in the background."

A Few Tips for Updating

- **One service at a time:** If you have multiple services, update one, test it, then move to the next
- **Check the logs:** After updating, run `docker-compose logs -f [service-name]` to see if anything broke
- **Don't panic:** If an update breaks something, you can always go back. Docker keeps old versions around until you explicitly delete them

What About Breaking Changes?

Sometimes a developer releases an update that changes how their service works. Maybe the interface looks different, or a setting moved. This is rare, but it happens.

The solution? Before updating, check the service's documentation or the comments on Docker Hub. Usually, someone will warn the community if an update requires extra steps.

Backups: Your Safety Net

Here's a scenario that happens to everyone: you're making changes to your server, something goes wrong, and–oops–your data is gone. Maybe you deleted the wrong folder. Maybe a drive failed. Maybe a software update didn't go well.

Without a backup, you've lost everything.

With a backup, you restore in minutes and move on with your life.

What Do You Actually Need to Back Up?

If you've been following this guide, you have a `docker-compose.yml` file that defines your services. This file is important–but it's also easy to recreate. The *real* precious stuff is your data:

- **Your media files:** Videos, music, photos
- **Your AdGuard statistics:** All those ad-blocking logs
- **Your Vaultwarden passwords:** Obviously important
- **Jellyfin library data:** Your watch history, user settings

In Docker terms, this is everything in your `volumes` section–the folders that live on your actual computer instead of inside the container.

The Simple Backup Method: Copy Your Folders

The most straightforward backup approach is to just copy your data folders to another location. You could:

- Copy them to an external hard drive
- Copy them to a different computer on your network
- Copy them to cloud storage (Dropbox, Google Drive, etc.)

Here's a simple bash script that backs up your Docker data folder:

```
#!/bin/bash
DATE=$(date +%Y-%m-%d)
tar -czf backup-$DATE.tar.gz /home/yourusername/docker-data
```

This creates a compressed file called `backup-2024-01-15.tar.gz` (or whatever today's date is) containing everything in your data folder.

The Slightly Fancier Method: Restic

If you want something more robust, there's a tool called **Restic** that does smart backups. Instead of copying everything every time, it remembers what changed and only backs up the differences. It's like how Google Photos knows you added three new pictures, so it only uploads those three instead of your entire photo library.

Installing Restic is just another Docker container:

```yaml
services:
  restic:
    image: restic/restic
    volumes:
      - ./backups:/backups
      - /path/to/your/docker-data:/data
    environment:
      - BACKUP_REPO=/backups
    command: backup /data
```

Set it up once, then schedule it to run automatically.

How Often Should You Back Up?

This depends on how often your data changes:

- **Vaultwarden (passwords):** Back up every time you add important new passwords, or at least weekly
- **Jellyfin (media):** You probably already have your original files somewhere, but back up metadata and watch history weekly
- **AdGuard:** Back up monthly or after making big configuration changes

---

Monitoring: Know What's Happening

You can't fix what you don't know is broken.

Here's the uncomfortable truth: if your server goes down right now, how would you know? Would you check it tomorrow? Next week? Would a friend trying to watch a movie on Jellyfin be the first to tell you?

That's where monitoring comes in.

What Is Monitoring?

Monitoring is simply keeping an eye on your services to make sure they're working. It's like having a smoke detector–hopefully it never goes off, but you'll be glad it's there if something catches fire.

Meet Uptime Kuma

Uptime Kuma is a self-hosted monitoring tool that's surprisingly fun to use. It basically asks your services "Are you alive?" every few seconds, and logs whether they respond.

When a service stops responding, Uptime Kuma can:

- Send you an email
- Send you a Telegram message

- Show you a notification
- Play a sound

Setting up Uptime Kuma takes about five minutes:

```
services:
  uptime-kuma:
    image: louislam/uptime-kuma
    container_name: uptime-kuma
    volumes:
      - ./uptime-kuma-data:/app/data
    ports:
      - "3001:3001"
    restart: unless-stopped
```

After this runs, open your browser and go to `http://[your-server-ip]:3001`. You'll set up your first account, and then you can add monitors.

Adding Your First Monitor

In the Uptime Kuma interface, click "Add New Monitor." Here's what you'll fill in:

- **Monitor Type:** HTTP(s)
- **URL:** `http://[your-server-ip]:[service-port]` (for example, `http://192.168.1.100:3000` for AdGuard)
- **Heartbeat Interval:** How often to check (every 60 seconds is fine for most services)
- **Timeout:** How long to wait before deciding it's down (30 seconds is good)

Now Uptime Kuma will check your service every minute. If it stops responding, you'll get notified.

What Should You Monitor?

At minimum, monitor:

- **AdGuard:** Because ad-blocking is your first service
- **Jellyfin:** Because you want to know when movies won't play
- **Vaultwarden:** Because password access matters
- **Your server itself:** Some people monitor whether the server is reachable at all

You can add as many as you want. There's no limit.

Bringing It All Together

Now you have three layers of protection:

1. **Updates:** Your services stay current with security fixes and new features
2. **Backups:** Your data survives hardware failures and mistakes
3. **Monitoring:** You know when something breaks

This is what "set and forget" actually means–not that you never touch your server, but that you've set up systems that run automatically and tell you when attention is needed.

You don't need to check your server every day. You just need to know that if something breaks, you'll find out quickly.

A Note on Maintenance Mindset

Here's the secret that experienced self-hosters know: **maintenance is not a chore–it's a habit.**

Once a month, you might:

- Check your backups ran successfully
- Look at Uptime Kuma to see if anything had issues
- Run `docker-compose pull` to get the latest updates
- Spend 15 minutes reading about new features in your favorite services

That's it. Fifteen minutes a month to keep everything running smoothly.

Do This Now

Let's put this chapter into practice right now.

Your exercise: Set up Uptime Kuma and add one monitor.

1. Add the Uptime Kuma container to your `docker-compose.yml` (or create a new file just for it)
2. Start it with `docker-compose up -d`
3. Open your browser to `http://[your-server-ip]:3001` and set up your account
4. Add one monitor for AdGuard Home (or whichever service you set up first)
5. Wait a minute and refresh the page–you should see it's "UP" with a green dot

That's it. You've just added monitoring to your server. Now if something breaks, you'll know.

When you're done, take a screenshot. You're officially a server administrator now–and server admins sleep better at night because they monitor their systems.

What's Next?

You made it. You now have:

- A running server with useful services
- Remote access so you can use your services anywhere
- Security (HTTPS, passwords)
- Backups to protect your data
- Monitoring to alert you to problems

That's a fully functional self-hosted setup. Congratulations!

In the final chapter, we'll talk about where you can go from here. Maybe you want to explore home automation with Home Assistant, or set up your own cloud storage with Nextcloud, or dive deeper into the Docker rabbit hole. Whatever calls to you, you now have the foundation to explore.

But for now, celebrate what you've built. You've joined a community of people who understand that having control over your technology is worth the effort.

Welcome to self-hosting.

Chapter 11: Where Do You Go From Here?

You've done something. You set impressive up a server from scratch, installed Docker, ran containers, got ads blocking across your network, streamed media from your own server, and even accessed it from outside your home. You learned about networking, security, and maintenance along the way.

That's not small stuff. Most people never get past "I should really learn to code someday." You're actually doing it.

But here's the thing: where you go next depends entirely on what *you* care about. Self-hosting isn't one path–it's dozens of paths that all start from the same place. This chapter helps you figure out which direction fits your life.

Finding Your Direction

Think back to Chapter 1, where you wrote down three things you wanted to achieve. That list still matters. The best next step is the one that solves a problem you actually have, not one that sounds cool on a YouTube video.

Ask yourself: What's the one frustration I have right now that technology could fix?

- Is it that my photos are scattered across devices?
- Do I want my home lights and thermostat to work together?
- Am I tired of typing the same repetitive tasks over and over?
- Do I want to run my own calendar and contacts without Big Tech?

Whatever your answer, there's a self-hosted tool for it. Let's look at the main paths.

Path 1: The Smart Home Path -> Home Assistant

If you want your lights, sensors, thermostats, and gadgets to talk to each other, Home Assistant is the answer. It's the most popular open-source smart home system in the world, and for good reason.

Home Assistant connects to thousands of devices–even ones that don't normally work together. That old thermostat? The smart bulbs you bought on sale? The motion sensor in the garage? Home Assistant brings them all into one place.

What it does: Gives you a dashboard to control everything in your home, automations that run on their own (like "turn off all lights when everyone leaves"), and works with almost every smart device brand.

Why it's great for you: You already have a server running. Installing Home Assistant takes about 10 minutes with Docker. You'll recognize the compose file

structure from Chapter 7.

Where to start: Check out the official Home Assistant website (home-assistant.io) and look for the Docker installation guide. The community is massive, so YouTube has excellent tutorials for beginners.

Path 2: The "My Own Cloud" Path -> Nextcloud

If you want to replace Google Drive, iCloud, or Dropbox with something you own, Nextcloud is the gold standard. It's like having your own private Google Workspace.

What it does: File storage and sync (like Dropbox), but also calendar, contacts, notes, video calls, and even document collaboration. You can install apps for almost anything–it's like a marketplace of useful tools.

Why it's great for you: Your server already has Docker. Nextcloud runs in a container just like everything else you've installed. You'll use volumes (remember those from Chapter 7?) to keep your files safe.

Where to start: The official Nextcloud Docker guide walks you through it. Be aware that Nextcloud needs a bit more setup than AdGuard–give yourself 30 minutes the first time. The documentation is good, and the subreddit r/Nextcloud is helpful if you get stuck.

Path 3: The Automation Path -> n8n

If you find yourself doing the same thing over and over–copying data between apps, sending the same notification, checking a website for updates–n8n (pronounced "n-eight-n") automates it.

What it does: Connects different services together. For example: "When I get an email with an attachment, save it to my Nextcloud and send me a text." Or: "Every morning, pull weather data and add it to my calendar." You build workflows by connecting blocks together, like digital LEGO.

Why it's great for you: n8n has a visual interface–you don't write code, you draw flows. But it's powerful enough that businesses use it too. You'll feel smart the moment your first automation runs on its own.

Where to start: The n8n website (n8n.io) has a cloud version, but you want the self-hosted version. Install it via Docker and start with one simple workflow, like forwarding a specific type of email to a folder.

Path 4: The Media Fortress Path -> More Media Tools

If Jellyfin got you excited, there's a whole world beyond it.

Plex: Similar to Jellyfin but with a more polished interface and optional paid features. Some people prefer it; some prefer Jellyfin. Try both.

Sonarr, Radarr, and Lidarr: These automate your media collection. Tell Sonarr which TV shows you want, and it finds and organizes them automatically. Radarr does the same for movies. Lidarr for music. They're like having a personal media manager that works while you sleep.

Jellyseerr: A request system for your media. Install this and your family or friends can request movies and shows. Radarr automatically grabs them. It's incredibly satisfying.

Where to start: Each of these runs beautifully in Docker. Search for "Trash Guides" online–they're the gold standard for how to set up these tools the right way.

Path 5: The Privacy Fortress Path -> More Security Tools

If locking down your digital life excites you more than automating it, here are some next-level tools.

Vaultwarden: You installed this in Chapter 7 as a password manager. But there's more: you can enable the Bitwarden browser extension, set up emergency access for family, and even share passwords securely with others.

Authy or Bitwarden Authenticator: Add two-factor authentication to your accounts. Self-hosting your 2FA codes is the ultimate privacy move.

Pi-hole: Think of it as AdGuard Home's bigger sibling. It does network-wide ad and tracker blocking at the DNS level, and has more customization for advanced users. You can even run Pi-hole and AdGuard together for maximum blocking.

Path 6: The "I Want to Learn More" Path

Maybe you're not sure what to build yet. You just know you want to get better at this. Here's how to level up.

Learn Linux properly: You've been using Ubuntu Server, but you now have a perfect lab to experiment. Mess around with the command line, learn about permissions, understand how processes work. The "Linux Basics" course on Linux Journey (linuxjourney.com) is free and excellent.

Try virtualization: Instead of running everything directly on your server, you can run virtual machines–computers within your computer. Proxmox is the most

popular choice for home users. It lets you run multiple independent systems on one piece of hardware.

Set up monitoring: Remember Uptime Kuma from Chapter 10? Pushover or Gotify can send you real alerts when things break. Build a dashboard that shows you everything about your server in one glance.

Join the community: The self-hosting subreddit (r/selfhosted) has over 400,000 members. The Home Assistant community is massive. Discord servers exist for nearly every major tool. Don't learn alone–ask questions, share what you've built, and learn from others.

Resources for Continued Learning

Here's where to go when you get stuck or want to learn more:

YouTube Channels: - *Techno Tim* - Excellent tutorials on Docker, Home Assistant, and infrastructure - *DB Tech* - Great for beginners, clear explanations - *Iron Geek* - Good for security-focused content - *Everything Smart Home* - Focused on Home Assistant

Websites: - *Awesome Selfhosted* (github.com/awesome-selfhosted/awesome-selfhosted) - A massive list of every self-hosted tool - *Linuxize* - Tutorials for common server tasks - *StackLinux* - Beginner-friendly guides

Communities: - r/selfhosted on Reddit - r/HomeAssistant - r/Docker - Discord servers for specific tools (Home Assistant, n8n, and others have official ones)

A Note on Going Deeper

As you add more services, your server will get more complex. That's fine–that's how you learn. But a few reminders from earlier chapters:

- Update regularly (Chapter 10)
- Check your backups
- Don't expose things to the internet unless you really need to
- When something breaks, Google the error message. Someone else has had the same problem.

The self-hosting community is incredibly generous with knowledge. Almost every problem you face has been solved and documented somewhere.

Your Turn: Do This Now

Here's your final exercise–the one that turns this book from something you read into something that changes your daily life:

Step 1: Read back through this chapter and the paths listed.

Step 2: Pick ONE path that excites you. Not five. One.

Step 3: Spend 15 minutes researching that specific tool. Search for "[tool name] Docker tutorial" or "[tool name] getting started."

Step 4: Install it this week. It doesn't have to be perfect. It just has to run.

That's it. You've already done the hardest part–you started. Everything from here is building on what you already know.

What's Next Is Up to You

Self-hosting isn't a destination. It's a continuous process of learning, building, breaking, and fixing. Some weeks you'll learn something new and feel like a wizard. Other weeks something will break at 2 AM and you'll question everything.

Both are part of the deal. And both are worth it.

You now have skills that most people never develop. You understand how the internet actually works under the hood. You can build tools that solve real problems in your life. You own your data and your infrastructure.

That's powerful. Use it well.

Go build something.