First Edition

HASHICORP VAULT FOR BEGINNERS

By Intekhab Rizvi

First Edition

Hashicorp Vault for beginner

By Intekhab Rizvi

I dedicated this book to my master of time (a.t.f.s) and his mother (s.a) – Without them, I am nothing.

About Book

If you've ever felt like Hashicorp Vault is powerful but... kinda intimidating - you're not alone. This book is here to fix that.

We'll walk you through Hashicorp Vault step by step, starting from scratch. No assumptions. No jargon without explanation. Just clear, real-world examples that work.

Whether you're a backend developer, DevOps engineer, or someone who wants to build more secure systems, this book was written with you in mind. And no, you don't need to be a Linux wizard. We'll be using Docker throughout, so you're good if you can run docker compose up.

Each chapter builds on the last. That means it's best to go in order, no skipping ahead. Trust the flow. You'll go from spinning up Vault for the first time, all the way to automating secret access in production-like scenarios.

Along the way, you'll:

- Understand how Vault works under the hood
- Store and retrieve secrets the right way
- Rotate database credentials on the fly
- Authenticate services using AppRole
- Use Vault Agent to make legacy apps play nice with secrets
- Encrypt/decrypt sensitive data using Vault's built-in crypto engine

All source code, configuration files, and Docker setups are available at our companion GitHub repo:

https://github.com/intekhabrizvi/vault-beginner

It’s all there. Tested, copy-pasteable, and ready to go.

Feel free to explore, fork, and ask questions. This is a hands-on book, and your curiosity is welcome.

So, crack open a terminal, grab a coffee, and build something secure together.

Meet the Author

Hi, I'm Intekhab Rizvi. I have over 20 years of experience building large-scale enterprise software platforms.

Academically, I hold a Master's degree in Computer Application. Professionally, I'm certified as a Hashicorp Vault Associate and Hashicorp Vault Operations Professional, and also have several AWS credentials, including Certified Solutions Architect and Security Specialist. You can find the rest of my certifications at http://intekhab.in

This book came from a real need for a simple, hands-on guide. I've often found that, while thorough, official documentation can be overwhelming and hard to follow for beginners. So I wrote the book I wish I had when I started years back.

My goal here is simple: make Vault approachable. Make it click. And if something doesn't make sense, reach out. I'm more than happy to help.

You can email me anytime at **me@intekhab.in**. Yes, I will check and reply personally.

Index

Chapter 1: Introduction to Hashicorp Vault

Hashicorp Vault is a tool built for a particular but increasingly critical problem: how do we securely manage secrets in modern software systems?

As software systems grow more distributed, cloud-based, and automated, the number of credentials, API keys, tokens, and passwords we need to manage grows. These secrets often end up hardcoded in source code, scattered in environment variables, or shared insecurely across teams. Vault provides a central, secure, and auditable way to manage and access secrets.

At its core, Vault acts like a digital vault for your sensitive information. But it's more than just a storage box with a lock. Vault is a dynamic secrets manager, meaning it can generate secrets on the fly, rotate them automatically, and revoke access when they're no longer needed. It can integrate with identity systems, enforce fine-grained access control, and encrypt application data without storing it.

But don't worry if all that sounds like a lot right now. That's precisely what this book is here for.

What You'll Learn

This book is a hands-on, practical guide designed to take you from zero to confident in Hashicorp Vault. We'll start with the basics, what Vault is, how to install it locally using Docker, and how to interact with it. As the chapters progress, we'll build a strong

foundation and then dive into more advanced topics like dynamic secrets, authentication backends, encryption as a service, and secure application integration.

Each chapter introduces key concepts in a structured way, followed by practical examples you can try on your machine. By the end of the book, you won't just know what Vault does, you'll understand how to apply it in real-world projects.

If you're a DevOps engineer, a backend developer, or a security-conscious builder of systems, you're in the right place. No deep Linux or command-line knowledge is required; we'll use Docker to simplify everything, ensuring Windows, macOS, and Linux users are equally supported.

Why Vault?

There are other secrets managers: AWS Secrets Manager, Azure Key Vault, and even Kubernetes Secrets. But Vault stands out for its flexibility. It's cloud-agnostic, open source, and built with complex infrastructure needs in mind. You can run it locally, in a datacenter, or the cloud. You control where your secrets live and who can access them.

Vault isn't just valuable for large-scale enterprise systems. It's just as relevant for small teams, hobby projects, or single-node services that need to manage secrets securely and cleanly.

> You'll also see callouts like this one throughout the book. These are tips, best practices, or warnings to help you avoid common mistakes. Please don't ignore them.

A Quick Word on Installation

We'll be running Vault locally using Docker. That means there's no need to install anything directly on your system or interact with raw binaries or config files unless you want to.

Vault can feel intimidating at first. It introduces a new way of thinking about secrets, authentication, and access control. But don't worry, you're not alone.

This book is written for real people building real systems. We'll go slow when needed, speed up when we can, and always ensure you understand what's going on before moving forward.

So take a deep breath. We've got you.

Please clone the repository linked below, which contains all the configurations, commands, and code samples used throughout this book. Access to this repository will make it easier for you to follow along with the material.

https://github.com/intekhabrizvi/vault-beginner/

Chapter 2: Installing Vault Using Docker

In this chapter, we'll launch Hashicorp Vault using Docker in a way that persists all Vault data even after stopping the container.

To make this hands-on experience smoother, we've already prepared the Vault configuration file and data directory for you. You don't need to write the configuration yourself; follow the steps, and you'll be up and running quickly.

Step 1: Clone Pre-bundled Repo

All the required configuration files, the Vault data directory structure, and chapter-wise example folders are already organized for you.

Before you continue, **please clone the following GitHub repository**, which contains everything you'll need as we progress through the book:

```
git clone https://github.com/intekhabrizvi/vault-beginner.git
cd vault-beginner
```

Step 2: Launching Vault with Persistent Storage

We'll run Vault using Docker and ensure it writes to a local folder (`vault-beginner/vault-data/file`) so the data survives across container restarts.

The pre-created configuration file is stored at `vault-beginner/vault-data/config/vault.hcl` location.

Use the following Docker command to start Vault:

```
docker run -d --rm \
  -p 8200:8200 \
  -v $(pwd)/vault-data/config:/vault/config \
  -v $(pwd)/vault-data/file:/vault/file \
  -v $(pwd)/vault-data/logs:/vault/logs \
  -v $(pwd):/home/vault/vault-beginner \
  --cap-add=IPC_LOCK \
  --name vault-dev \
    hashicorp/vault:latest server
```

Note: If you're on Windows, make sure to adjust the `"$PWD"` path accordingly to point to the absolute path of the `vault-beginner` repo on your machine.

Explanation by Segment

Part	Purpose
`docker run`	This starts a new container instance from the specified Docker image.
`-d`	Runs the container in detached mode, meaning it runs in the background.
`--rm`	Automatically removes the container when it stops. This is useful for cleanup. It deletes the container (not the data) once stopped.
`-p 8200:8200`	This maps port `8200` on your host to port `8200` inside the container. Vault uses port 8200 for its HTTP API and UI.
`-v $(pwd)/vault-`	Mounts the **Vault config folder** from your host into

Part	Purpose
`data/config:/vault/config`	the container at `/vault/config`. This is where your `vault.hcl` file lives.
`-v $(pwd)/vault-data/file:/vault/file`	Mounts the **data storage folder** for Vault's persistent backend. This is where Vault saves secrets, tokens, leases, etc.
`-v $(pwd)/vault-data/logs:/vault/logs`	It mounts a log directory so that Vault's logs can be persisted on by your host for inspection.
`-v $(pwd):/home/vault/vault-beginner`	Mounts the **vault-beginner** folder we just cloned inside the vault container. This will help you to execute the commands quickly.
`--cap-add=IPC_LOCK`	Grants the container permission to lock memory using `mlock`. This prevents sensitive data from being swapped to disk. It's a critical security practice.
`--name vault-dev`	Assigns a name to your container, making it easier to reference later with `docker exec`, `docker logs`, etc.
`hashicorp/vault:latest`	Specifies the image to use. You're using the latest official Hashicorp Vault Docker image.
`server`	Tells Vault to run in **server mode** (as opposed to client mode).

Step 3: Attach to the Vault Container

To interact with Vault CLI inside the container:

```
docker exec -it vault-dev /bin/sh
```

Now you're inside the Vault container.

Step 4: Set the `VAULT_ADDR` Environment Variable

After the container starts, you must set the `VAULT_ADDR` environment variable. This environment variable is used by Vault clients (including the CLI) to point to the correct Vault server address. To set the `VAULT_ADDR` variable, run the following command **inside the container.**

```
export VAULT_ADDR=http://localhost:8200
```

Important Notes

1. **Persistence**: The data is persisted because of the volume mounts. The data will survive in the mounted host directories even if the container is deleted (--rm).
2. **Memory Lock**: `--cap-add=IPC_LOCK` enables `mlock`, but you must still set `disable_mlock = false` in your Vault config (`vault.hcl`) to allow it fully.
3. **Configuration**: The config file (`vault.hcl`) should be placed in the `vault-data/config` directory, and must define the storage backend (e.g., file) and the listener.
4. **Accessing Logs**: You can view Vault logs from the mounted folder `vault-data/logs` or use `docker logs vault-dev`.
5. **Stopping the Container**: Since `--rm` is used, once you stop the container, it will be deleted. But the data will remain on your host via the `vault-data/file` folder.

What's Next?

In the next chapter, we'll walk through how to initialize and unseal Vault from inside the container. We'll also explain the purpose of the unseal keys and root token and how Vault handles encryption from the moment you start using it.

Let's move forward!

Chapter 3: Vault Initialization and Logging In

Now that Vault is installed and running locally using Docker, it's time to bring it to life.

In this chapter, we'll introduce the `vault operator init` and `vault login` commands. You'll learn what it means to initialize Vault, how to unseal it, and how to authenticate. We'll also briefly touch on Vault tokens, what they are, and why they matter without diving too deep (we'll explore them thoroughly in a later chapter).

Wait, Why Does Vault Need to Be "Initialized"?

When Vault starts up for the first time, it's in a sealed state, thinking of it as a locked box with a self-destructive mechanism inside. Before anyone can use it, Vault must be **initialized** and **unsealed**.

Why? Because Vault doesn't want to take chances. It protects your secrets with an encryption key that doesn't exist until you run `vault operator init`. That command generates the cryptographic foundation Vault needs to operate, including the keys to unlock its secrets.

Once initialized, you can authenticate using a **token**, Vault's way of saying, "Yes, you have permission."

Before you begin: Ensure you have completed all steps in Chapter 2 and are still connected to the Vault container. If not, please revisit and execute the setup commands from that chapter first.

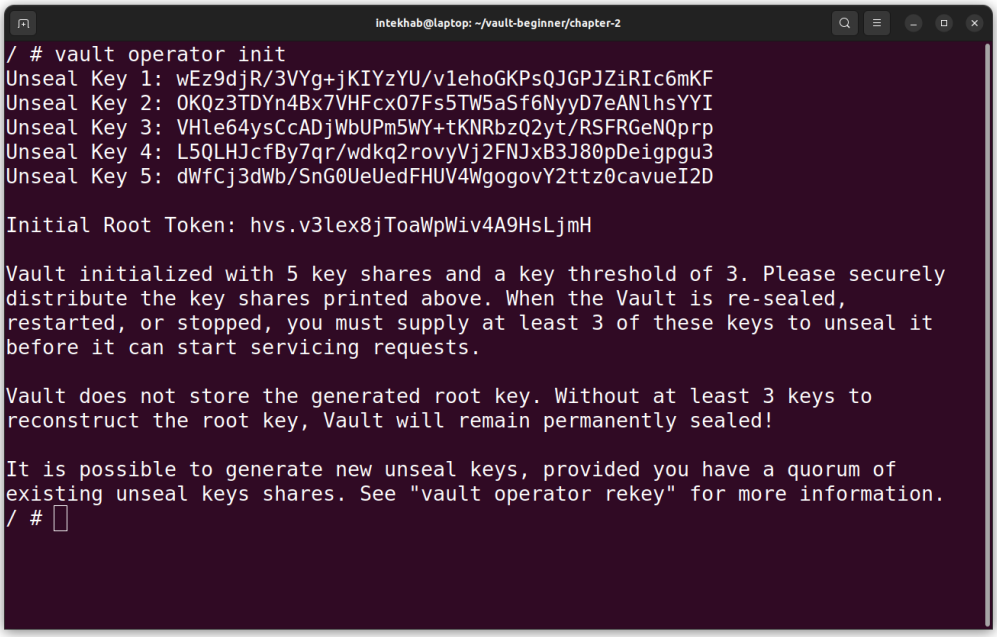
All commands in this section must be run from inside the Vault container shell.

Step 1: Initialize Vault

Run this command in your terminal:

```
vault operator init
```

You'll see output that looks like this:



Don't forget: Make sure to save the unseal keys and root token securely—you'll need them throughout this book. If you lose them, you must delete the `vault-data` folder and reinitialize Vault from scratch.

Vault uses a mechanism called **Shamir's Secret Sharing** to generate multiple **unseal keys**. These keys are required to "unlock"

the vault. By default, Vault requires **at least 3 of the 5** unseal keys to unseal it.

Yes, it's paranoid in a good way.

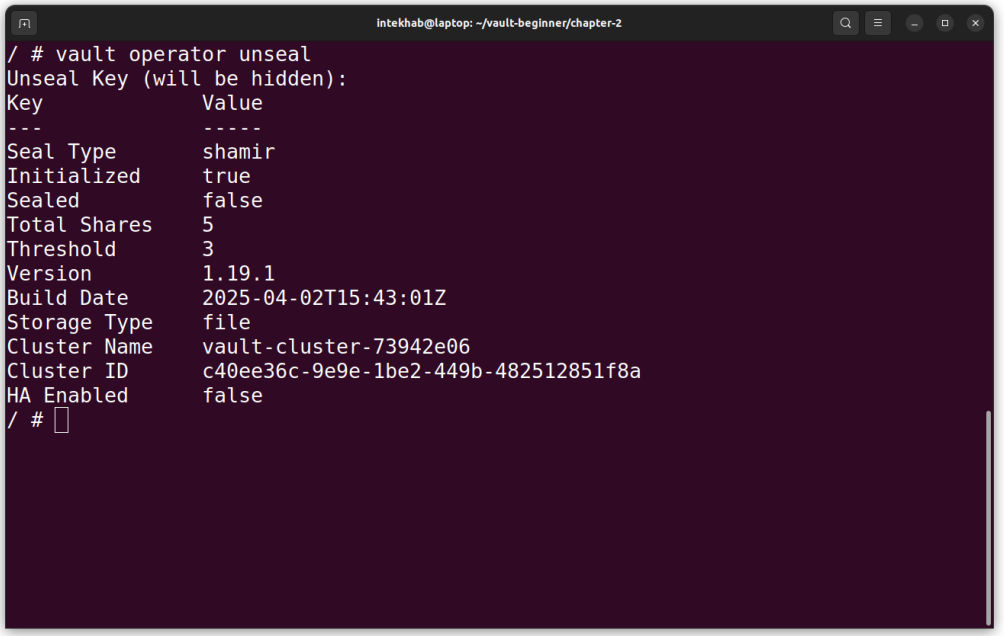
Step 2: Unsealing the Vault

To unseal Vault, run this command three times using three of the unseal keys:

```
vault operator unseal
```

Paste one key at a time when prompted.

After the third key, Vault becomes unsealed and ready for use.



Step 3: Logging In

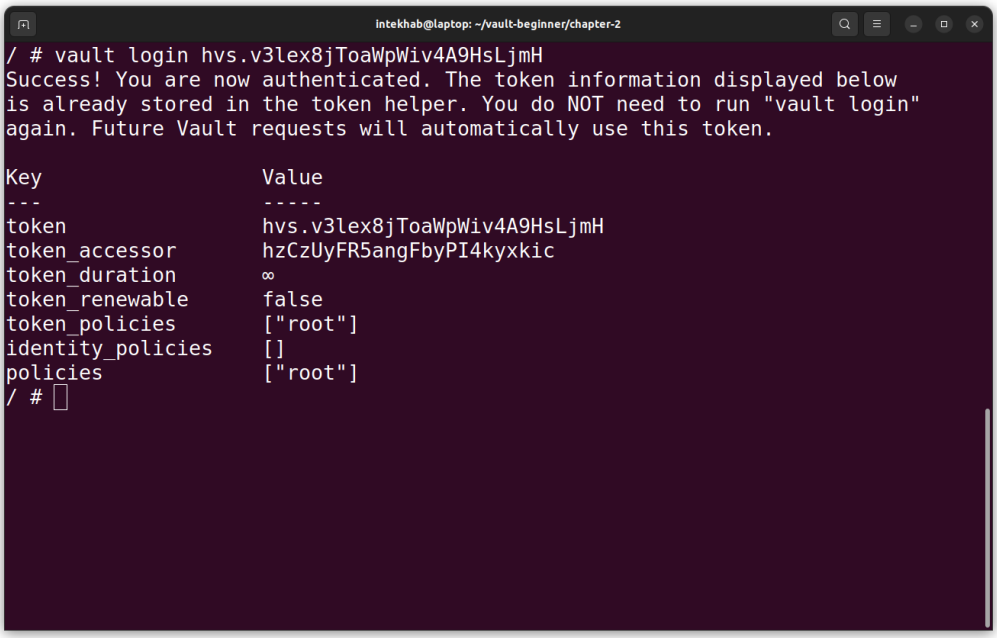
Once Vault is initialised and unsealed, you need to **authenticate**.

Use the root token generated during initialisation:

```
vault login hvs.abc123xyz456
```

(Replace `s.abc123xyz456` with the root token you received when performing the vault operator init.)

If everything goes well, you'll see:



You're in.

A Quick Word About Tokens

When you log in to Vault, you get a **token,** a digital passport that proves who you are and what you're allowed to do.

Tokens are central to how Vault handles identity and access. There are root tokens (all-access), regular tokens (limited access), renewable tokens (for long-lived services), and more. You can

revoke, rotate, and restrict them, and yes, we'll dive into all of that in a dedicated chapter later.

But for now, remember this:

> N**o token, no entry.**

Recap

- You **initialized Vault** using `vault operator init`.
- You **unsealed it** with three out of five unseal keys.
- You **authenticated** using the root token.
- And you got a first glimpse into Vault's token-based access control.

From here on, you're ready to start interacting with Vault, storing secrets, configuring access, and building secure workflows.

Chapter 4: Understanding Shamir's Secret Sharing and Rekeying Vault

In the last chapter, we briefly touched on Vault's initialization process and the concept of unseal keys. Now it's time to dig deeper.

This chapter explains the role of Shamir's Secret Sharing, the meaning and significance of key shares, how to rekey the Vault, and what happens when things go wrong, like losing too many key shares. We'll also address a common concern: Does the person initializing Vault see all the keys? (Spoiler: there's a way to avoid that.)

What is Shamir's Secret Sharing?

Hashicorp Vault uses a cryptographic algorithm called **Shamir's Secret Sharing** to protect its master key. Instead of storing one master key (a single point of failure), Shamir's algorithm **splits the key into multiple pieces**, known as **key shares**.

These shares are distributed among multiple trusted parties. To reconstruct the master key (and unseal the Vault), only a **subset** of those shares is needed, not all of them.

By default:

- Vault creates **five key shares**
- The **threshold to unseal** is set to **3 shares**

This means any 3 of the five key holders can come together to unseal the Vault. This is both secure and practical, and it avoids a single person becoming a bottleneck while ensuring no one can unseal the Vault alone.

Why Use Key Shares at All?

Because **trust is not binary**.

With Shamir's Secret Sharing, no single person holds all the power. It also lets you distribute trust across a team, organization, or countries if needed.

This becomes especially valuable in enterprise or compliance-driven environments where:

- Shared custody is a regulatory requirement
- Insider threat mitigation is essential
- Multi-party approvals are part of the access policy

But Wait, Doesn't the Person Who Initializes the Vault See All the Keys?

Yes, when you run vault operator init, all the unseal keys are printed to your screen by default. Whoever runs the command sees every key. This is risky. You're relying on that one person to distribute the keys securely, not screenshot them, and not keep copies.

That's where **PGP encryption** comes in.

Using PGP to Distribute Key Shares Securely

You can prevent the initializer from seeing the raw unseal keys by providing **PGP public keys** during initialization.

Vault will:

- Encrypt each key share with a corresponding PGP key
- Return only **the encrypted blobs**
- So only the intended key holders who have the matching private keys can decrypt and access their unseal key

Here's how:

```
vault operator init \
  -key-shares=5 \
  -key-threshold=3 \
  -pgp-
keys="pgp1.asc,pgp2.asc,pgp3.asc,pgp4.asc,pgp5.asc"
```

Each file (`pgp1.asc`, etc.) should contain the **ASCII-armoured public key** for one of your trusted operators.

> The creation of a GPG public key is covered in detail in Chapter 27.

The output will look something like this:

```
Unseal Key 1:
vault::pgp::TUVTVEVSU0VDUkVUV0lUSFZBVUxURk9STk8x...
Unseal Key 2:
vault::pgp::QUJDREVGR0hJSktMTU5PUFFSU1RVVldY...
```

Only the intended person can decrypt their share using their **private PGP key**, like this:

```
echo "<key>" | base64 -d | gpg -d
```

This makes the initialization process **zero-trust** for the initializer. They never see the actual unsealed keys.

Customising Key Shares and Thresholds

You can customize the number of shares and the unseal threshold with or without PGP. Example:

```
vault operator init -key-shares=7 -key-threshold=4
```

You can choose any numbers that fit your organization's security needs, as long as:

- `threshold <= shares`
- You have a secure distribution plan for the shares

What If You Lose Key Shares?

This is where it gets serious.

You are permanently locked out of the Vault if you lose more key shares than your threshold allows.
There is **no backdoor**, no "reset," no way to recover the Vault without enough shares.

For example: with `-key-shares=5 -key-threshold=3`, you're done if you lose three or more keys.

That's why it's essential to:

- **Back up unseal keys securely**
- Store them in encrypted vaults, HSMs, or offline media

- Use **PGP encryption during init** to avoid human errors

Rekeying Vault: Why and How

You may need to rotate unseal keys:

- A key holder left the team
- You suspect a compromise
- You're rotating keys on a compliance schedule

To rotate (rekey) unseal keys:

```
#init the rekey process
vault operator rekey -init -key-shares=5 -key-
threshold=3
vault operator rekey
```

Then, provide the current unseal keys (up to threshold) to complete the process.

Vault will generate **new keys**, and the old ones will stop working. Distribute the new ones securely.

You can also **rekey using PGP** the same way you initialized:

```
vault operator rekey -init \
  -key-shares=5 \
  -key-threshold=3 \
  -pgp-
keys="pgp1.asc,pgp2.asc,pgp3.asc,pgp4.asc,pgp5.asc"
```

When Rekeying Won't Work

- **Too many keys lost:** Can't gather the threshold = no rekey.

- **Vault is sealed and can't be unsealed:** No rekey until the Vault is unsealed.
- **Dev Mode:** Rekeying is disabled and irrelevant in `dev` mode.

What If You Lose the Minimum Threshold?

Vault will become **inaccessible forever**.

There is no recovery path if you lose more keys than the threshold allows. No master override. No root access. Vault isn't joking around it prioritizes **security and confidentiality over convenience**.

Recap

- Shamir's Secret Sharing splits the master key into safe, distributed key shares.
- Use `-key-shares` and `-key-threshold` to customise how many are created and required.
- Losing more than the threshold of unseal keys is unrecoverable.
- Use **PGP encryption** during initialization to prevent the initializer from seeing unseal keys.
- Rekeying lets you rotate unseal keys securely if you still have enough to meet the threshold.

Vault's security model is airtight, but you must handle the keys responsibly. In the next chapter, we'll dive into Vault's first real use case: storing secrets using the Key-Value secrets engine.

Chapter 5: Storing Your First Secret

Up until now, we've danced around Vault. You installed it, initialized it, logged in… but what good is a vault if it doesn't store secrets?

In this chapter, you'll:

- Log in using the CLI
- Enable the **KV (Key-Value)** secrets engine
- Store your first secret
- Retrieve it
- Delete it

Now, let's attach to the Vault container using the following command:

```
docker exec -it vault-dev /bin/sh
```

Step 1: Logging In to Vault

After initializing Vault, you receive a **root token,** a powerful key that gives full access.

Use it to log in via CLI:

```
vault login <your-root-token>
```

If successful, Vault says:

```
Success! You are now authenticated.
```

From now on, any Vault commands you run will use this authenticated session.

Step 2: Enabling the KV Secrets Engine

Vault doesn't store secrets by default. You must first enable a **secrets engine**, a logical backend for secret storage.

We'll enable the basic **KV (Key-Value)** engine. This version is **KV v1,** a simple key-value store with no versioning.

```
vault secrets enable -path=secret kv
```

This mounts the secrets engine at the path `secret/`. That means any secrets stored under this engine will live under the prefix `secret/`.

What's a "path"?
Vault organizes secrets into paths, think of them like folders. `secret/myapp` is one such path. In a later chapter, we'll dive deeper into path structures and access control.
What about KV v2?
KV v2 is the versioned version of this engine. You can store multiple versions of secrets, roll them back, and see edit history. We'll explore KV v2 in a later chapter. For now, we're sticking to the simpler KV v1 for clarity.

Step 3: Storing a Secret (KV Put)

Let's store a simple username and password.

```
vault kv put secret/myapp username=admin
password="S3cr3t!"
```

This creates a secret at the path `secret/myapp` with two fields: `username` and `password`.

Expected output:

```
Success! Data written to: secret/myapp
```

No versioning here if you overwrite the secret, the old one is gone.

A**dvanced Options:**
Vault supports multiple flags like `-mount`, `-cas`, or `-format`, and you can also structure your values using JSON. We'll explore these options later when we go deeper into customization.

Step 4: Retrieving a Secret (KV Get)

To read the secret back:

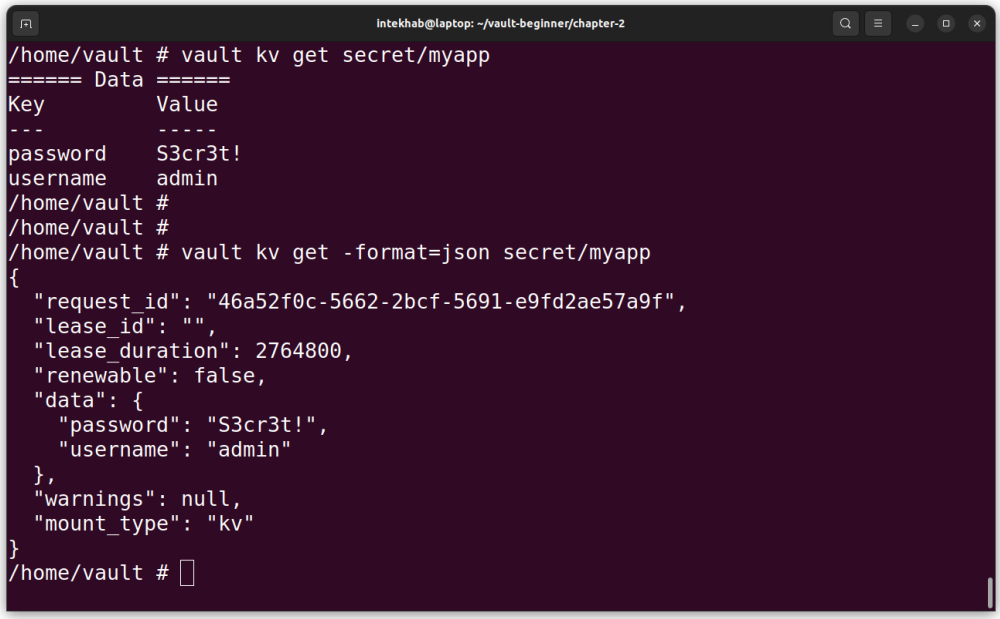
```
vault kv get secret/myapp
```

You'll see:

```
====== Data ======
Key         Value
---         -----
password    S3cr3t!
username    admin
```

You can also fetch raw JSON, and the output will be like below

```
vault kv get -format=json secret/myapp
```

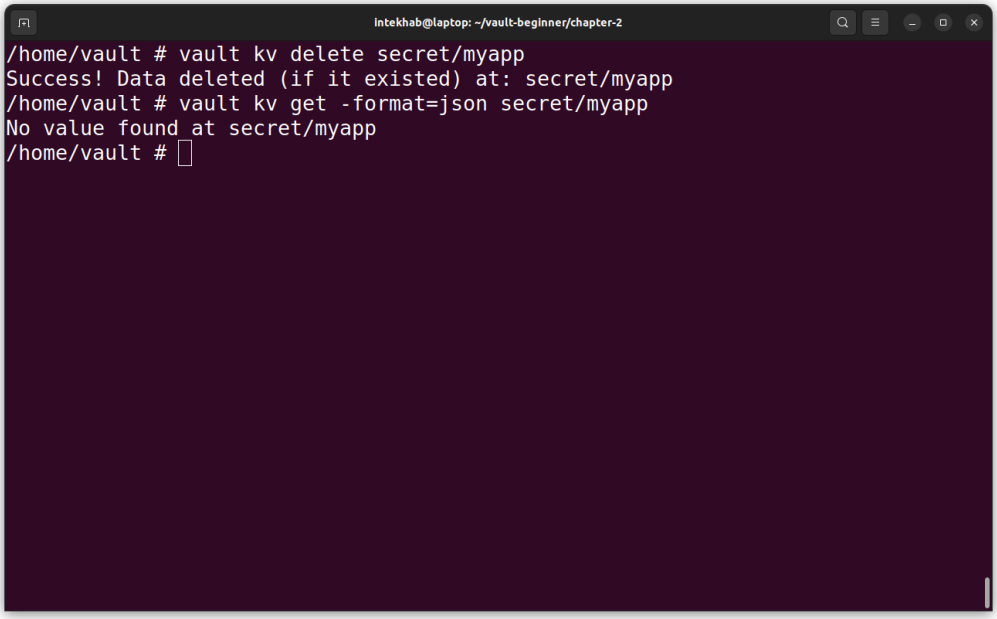


Step 5: Deleting a Secret (KV Delete)

To permanently delete a secret from KV v1:

```
vault kv delete secret/myapp
```

There’s no trash bin or versioning here; once it’s gone, it’s gone.



> In KV v2, `delete` only marks secrets as deleted, they can still be recovered. However, in KV v1, deletion is permanent. Choose carefully!

Let's write the data back into Vault, as we'll need it in the upcoming chapter:

```
vault kv put secret/myapp username=admin
password="S3cr3t!"
```

This ensures that the necessary secrets are available when we reference them later.

Recap

You just completed your first real Vault workflow:

- Logged in using the root token

- Enabled the KV (v1) secrets engine
- Stored a key-value pair
- Retrieved it
- Deleted it

You also got your first taste of Vault paths and engines, two fundamental concepts we'll build on in upcoming chapters.

Chapter 6: KV v2 Versioned Secrets

In our earlier chapters, we used the KV (Key-Value) secrets engine in its basic form, **KV v1**. Simple. Fast. No versioning. You put a secret in, you get it out. That's it.

But what if you accidentally overwrite a secret? Or delete something important? Or want to know what changed and when?

That's where **KV v2** comes in.

What is KV v2?

KV v2 is a versioned key-value store. It gives you:

- **Version history** for secrets
- **Soft deletes** (you can undelete!)
- **Metadata tracking** (who, when, what)
- And a little peace of mind

It's the "undo" button for your secrets.

Let's attach to the Vault container and log in using the root token.

Enabling KV v2

Just like KV v1, you enable KV v2 at a path. The difference? You need to tell Vault to use the **versioned** option explicitly.

```
vault secrets enable -path=kvv2 kv-v2
```

This creates a new mount point at `kvv2/` using the KV v2 engine.

> Note: You can call this path anything (like `secrets/`, `apps/`, etc.). Just remember: the path is part of the API. More on paths in the next section.

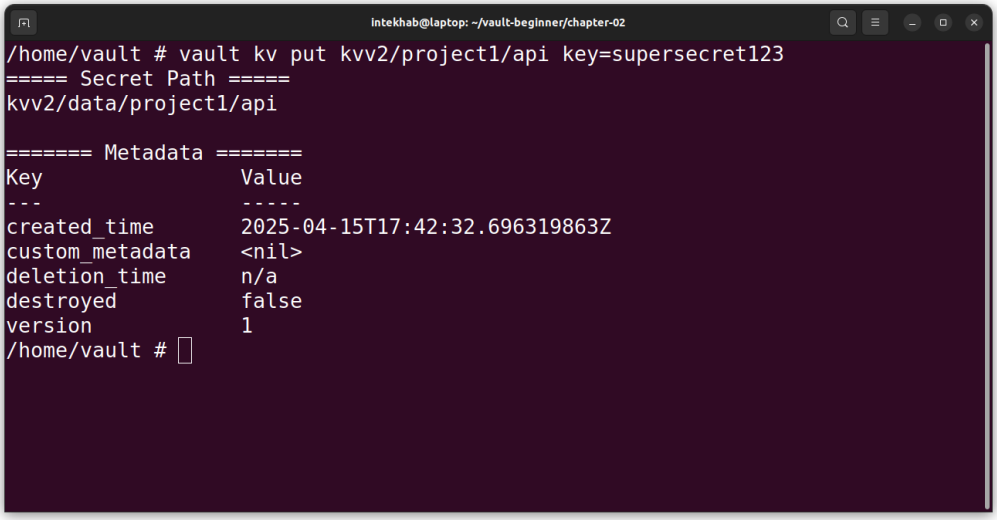
Putting Secrets into KV v2

Unlike KV v1, writing to KV v2 requires a slightly different path `/data/` **is required**.

```
vault kv put kvv2/project1/api key=supersecret123
```

This:

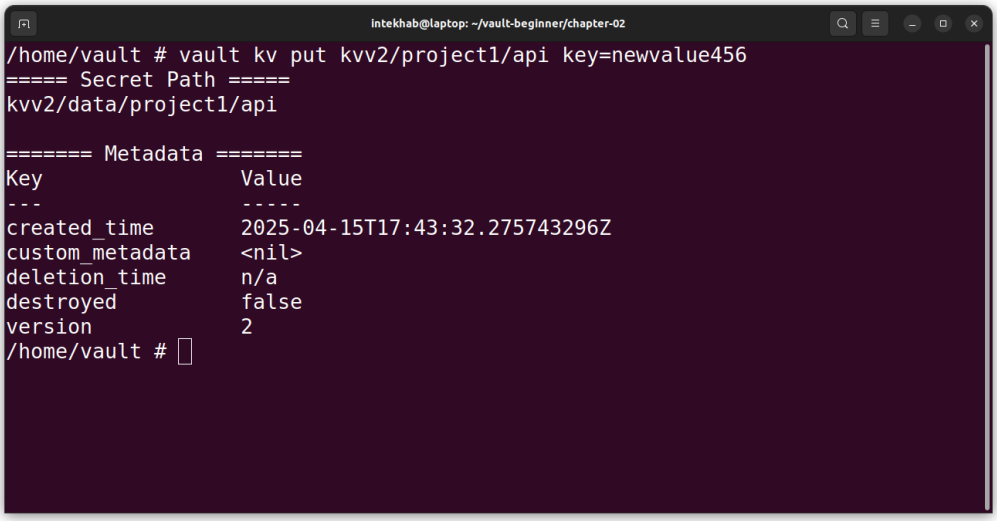
- Stores a secret at the logical path `kvv2/project1/api`
- Automatically saves it as **version 1**



If you do it again with new data:

```
vault kv put kvv2/project1/api key=newvalue456
```

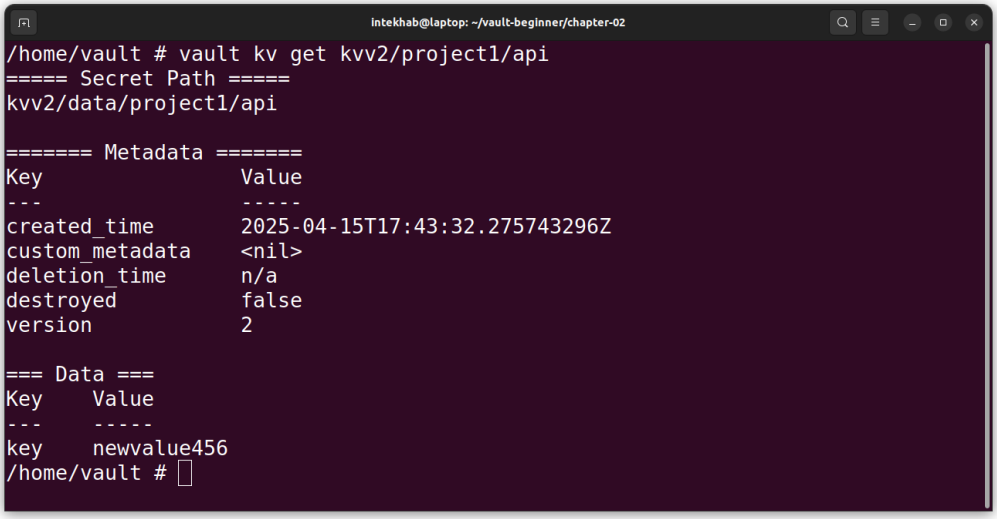
Boom, that’s version 2. Vault keeps the old one in the background.



Reading a Specific Version

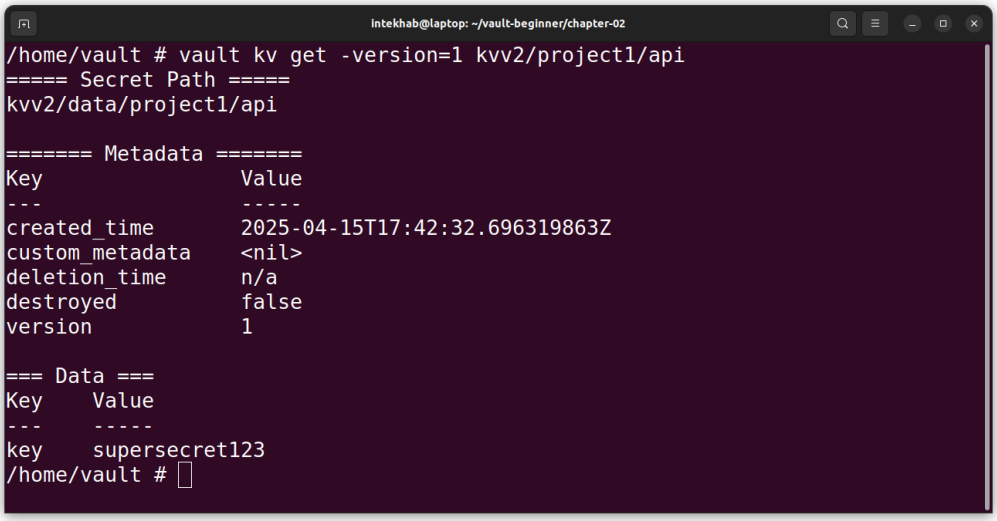
By default, Vault gives you the latest version:

```
vault kv get kvv2/project1/api
```



Want to see version 1?

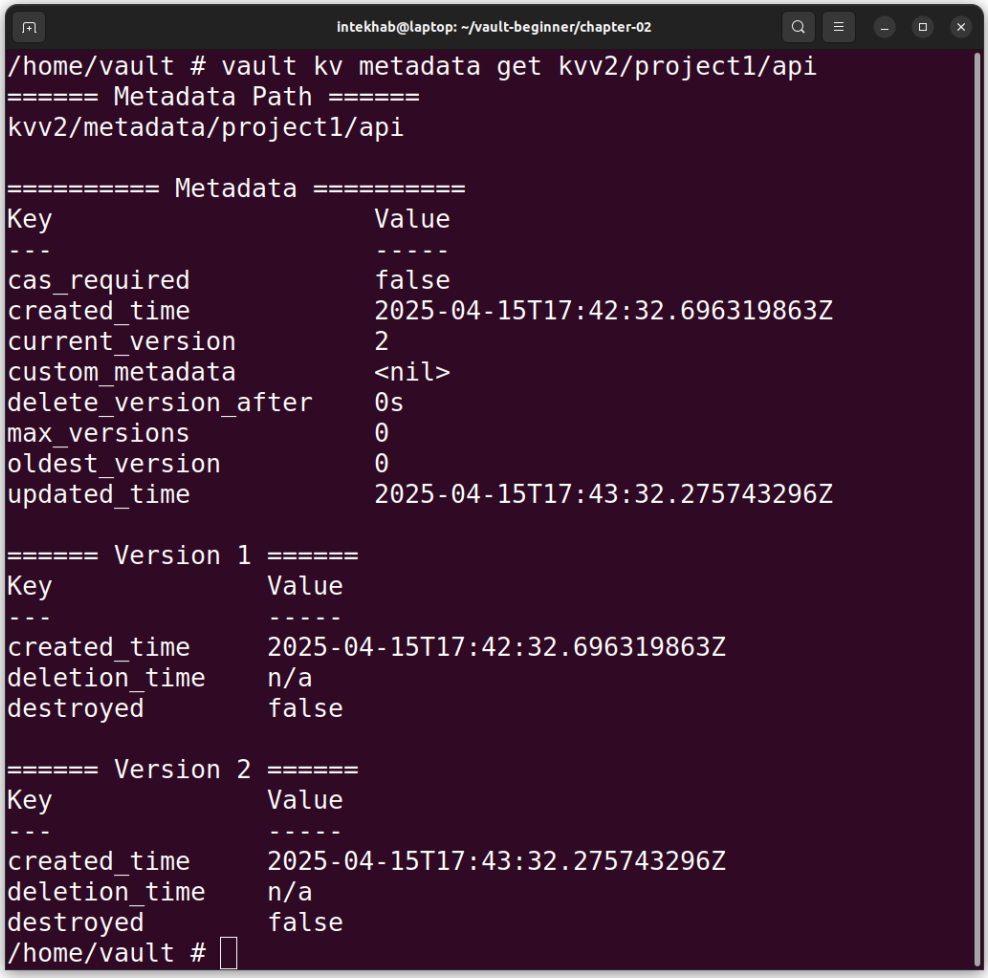
```
vault kv get -version=1 kvv2/project1/api
```



You can even inspect metadata:

```
vault kv metadata get kvv2/project1/api
```

This shows all versions, deletion flags, and timestamps.

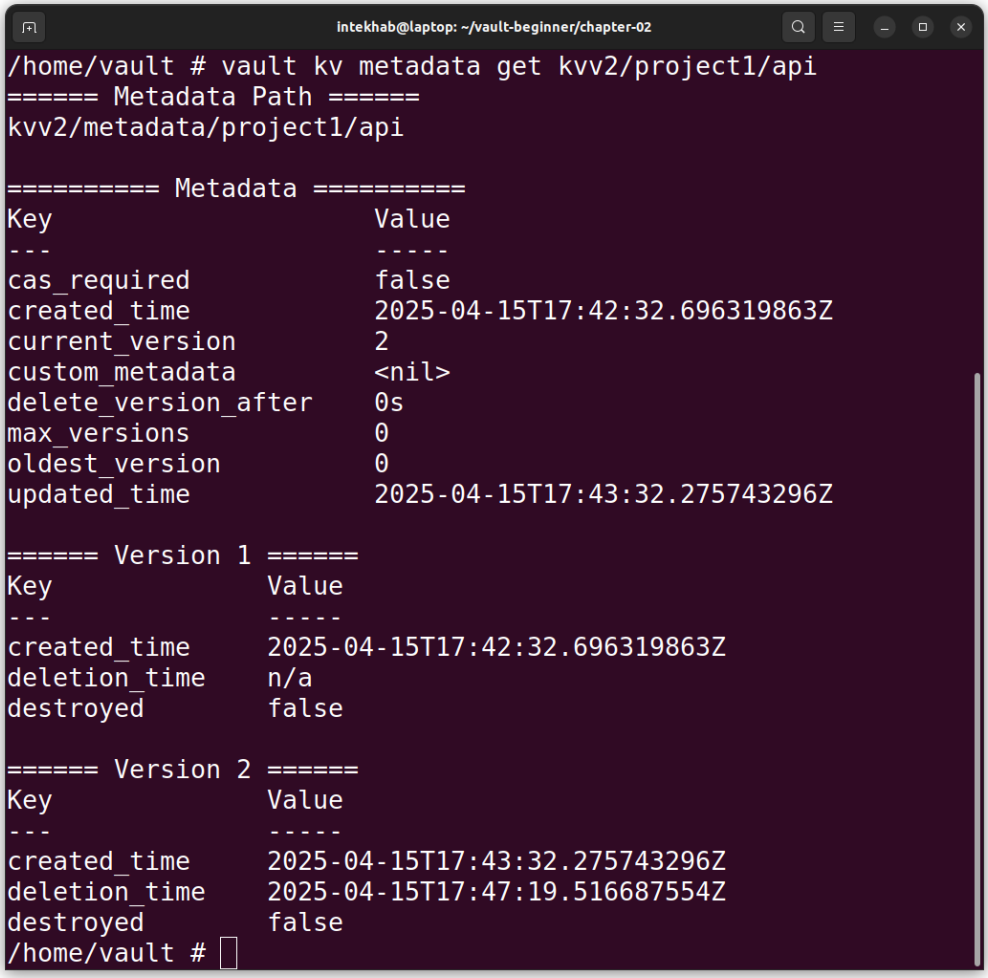


Deleting and Undeleting

Want to delete a version?

```
vault kv delete kvv2/project1/api
```

Note: This is a **soft delete;** it only marks the latest version as deleted.



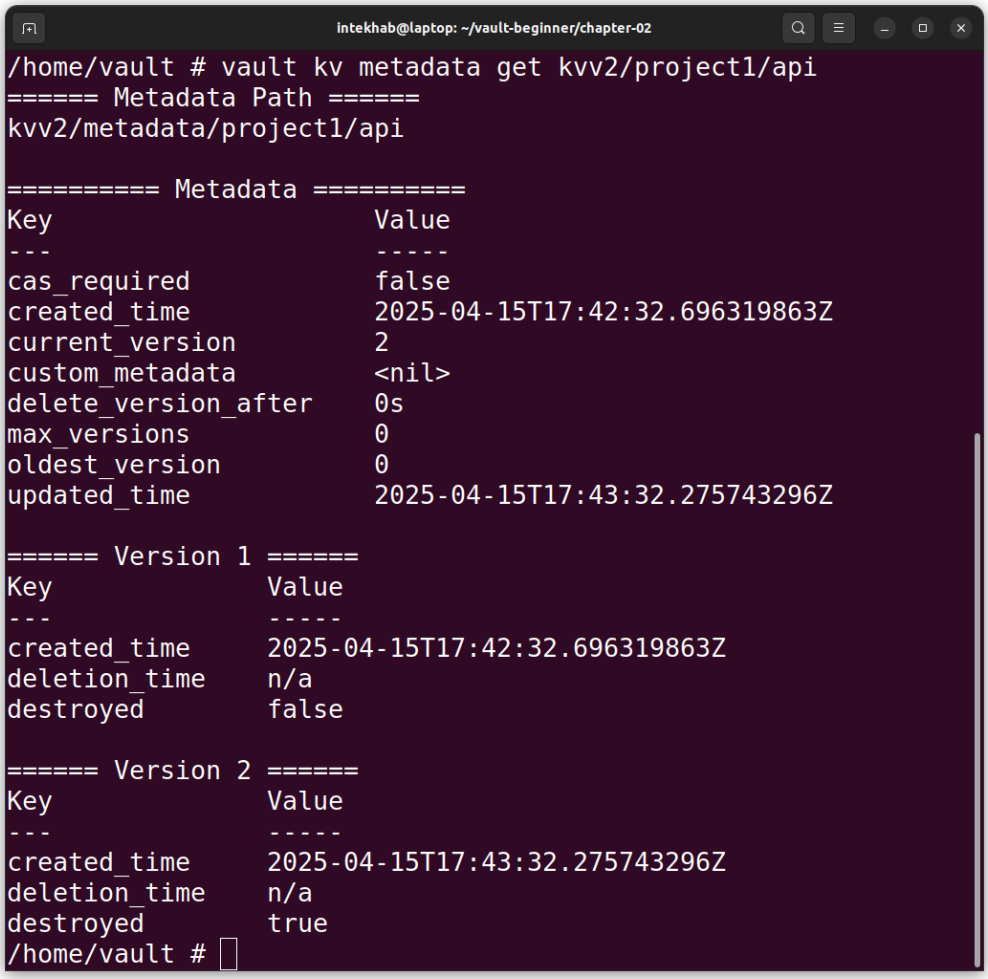
Or maybe you panicked and want to bring a deleted version back?

```
vault kv undelete -versions=2 kvv2/project1/api
```

Yes, Vault lets you undo a delete. Just like Ctrl+z.

To wipe it: That deletes version 2 permanently.

```
vault kv destroy -versions=2 kvv2/project1/api
```



Let's write the data back into Vault, as we'll need it in the upcoming chapter:

```
vault kv put kvv2/project1/api key=newvalue456
```

This ensures that the necessary secrets are available when we reference them later.

What's This "/data/" and "/metadata/" Stuff?

With KV v2, there are two APIs under the hood:

- `/data/`: For your actual secrets
- `/metadata/`: For managing and inspecting versions

This means you need to know which part you're talking to when using the CLI or API**.**

You can think of it like this:

Path Segment	What it Does
`/data/`	Put/get secrets (actual key-values)
`/metadata/`	List versions, delete metadata, etc.

We'll unpack the concept of **paths** more in the next chapter. They're foundational to how Vault works.

Quick Comparison: KV v1 vs KV v2

Feature	KV v1	KV v2
Versioning	No	Yes
Undelete	No	Yes
Metadata tracking	No	Yes
Extra path logic	Simple	Has `/data/`, `/metadata/`

What's Next?

Now that you understand **KV v2** and have your secrets version-controlled like a pro, it’s time to dig deeper into Vault’s architecture.

In the next chapter, we’ll explore how Vault organizes and manages data internally, and why that structure matters more than you might think.

Chapter 7: Secrets Engine and Paths

Vault isn't just a tool for storing secrets. It's a flexible platform that supports multiple types of secrets from passwords and tokens to cloud credentials, certificates, encryption keys, and even dynamic access to infrastructure.

At the center of all this magic are **Secrets Engines** and **Paths**.

Let's unpack both.

Secrets Engines: Not Just One Vault, But Many

A **Secrets Engine** is a backend plugin responsible for a specific type of secret.

Some examples:

Engine	What it stores or does
`kv`	Stores static key-value secrets
`aws`	Dynamically generates AWS access credentials
`database`	Generates database usernames/passwords on the fly
`transit`	Performs cryptographic operations (sign, encrypt)
`pkI`	Issues and manages TLS certificates
`identity`	Manages entities and groups inside Vault

In other words Vault isn't just one vault. It's many vaults under one interface, each specialized for a particular type of secret or operation.

> Think of each secrets engine as a plugin. When you enable one, you're telling Vault, "I want to store this kind of data here."

Enabling Secrets Engines

You enable engines like this:

```
vault secrets enable -path=mykv kv
```

This mounts the KV engine at `mykv/`.

You can have multiple mounts of the same engine, each with different purposes:

```
vault secrets enable -path=dev-kv kv
vault secrets enable -path=prod-kv kv
```

Each engine is isolated secrets under `dev-kv/` are completely separate from `prod-kv/`.

Paths: The Filesystem of Vault

Secrets live at **paths**, and paths are how you organize and control access to data.

For example:

- `secret/db/creds` might store static credentials
- `aws/creds/dev-role` might generate dynamic AWS keys
- `transit/keys/myapp` might contain encryption keys

You can structure paths any way you like. Just like folders on a hard drive.

But paths are **more than just location** they're also the foundation for:

- **Access Control** (Who can access what)
- **Auditing** (Who accessed what)
- **Policies** (Rules applied per path)

We'll dive into policies soon but remember: **Vault security is path-based**.

Secrets Engine Types: A Quick Classification

Vault secrets engines can be broadly grouped:

Category	Engines
Static storage	`kv` (v1 and v2)
Dynamic credentials	`aws`, `gcp`, `database`, `rabbitmq`, etc.
Encryption & signing	`transit`, `pkI`, `ssh`
Identity & auth	`identity`
Custom / community	Third-party plugins

Some engines support dynamic secrets, meaning Vault generates the secret when you request it and automatically revokes it later.

This is one of Vault's superpowers, never having to store or rotate secrets manually.

Why Paths and Engines Matter

- **Paths** define where secrets live, how they're organized, and how they're accessed.
- **Secrets Engines** define how those secrets behave and what they're for.

Understanding this structure helps you:

- Avoid clutter and chaos in large Vault deployments
- Secure your environment with precision
- Build automation around dynamic secret generation

What's Next?

You've now seen the structure beneath Vault, and why it's so much more than a key-value store.

In the next chapter, we'll learn how to define **access policies**, so your Vault isn't a free-for-all.

We'll write our first policy and assign it to a token. After all, what good is a vault if anyone can open every drawer?

Let's lock it down.

Chapter 8: Authentication Methods – Giving Others Access

Until now, you've been interacting with Vault using the root token. And that’s fine for learning. But in the real world?

Root tokens are like nuclear launch codes. Powerful, dangerous, and best kept far, far away from everyday use.

So how do we let others (humans, apps, machines) access Vault, safely?

Welcome to **Auth Methods**.

What is an Auth Method?

An **Auth Method** is how a user or application proves its identity to Vault.

Vault doesn’t have built-in users. It delegates identity verification to these pluggable methods?????like LDAP, GitHub, AWS IAM, Kubernetes, or good old `username/password`.

Once authenticated, Vault returns a **token** with **policies** attached. That token defines what that entity is allowed to do.

> Think of an auth method as the door, and the token as the keycard that Vault gives you once it knows who you are.

Why Use Auth Methods?

So far, only you, the Vault admin had the root token. But now it’s time to share the Vault love:

- Your team needs access
- Your apps need secrets
- Your CI/CD pipelines need short-lived credentials

But **you should never share the root token. Ever.**

Auth methods let you:

- Give access **without** giving away sensitive tokens
- Control **who can do what** using policies
- Audit, rotate, and revoke access cleanly

Enabling the Userpass Auth Method

Let's start with the most basic and easy-to-understand method: **userpass**.

This is Vault's built-in `username/password` login system. Great for quick demos or small teams.

Enable the userpass method:

```
vault auth enable userpass
```

You'll see:

```
Success! Enabled userpass auth method at: userpass/
```

Vault just mounted this method at the `userpass/` path.

> You can mount auth methods at any custom path (like `dev-login/` or `team-a/`). But for now, let's keep it simple.

Create a User

Let's create a user called **Alice**. She'll use a username/password combo to log in.

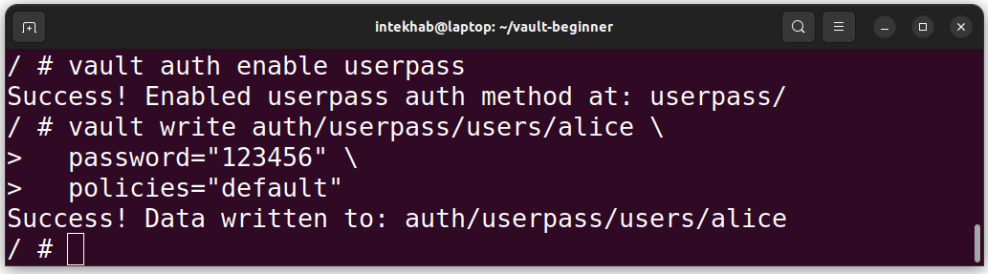
```
vault write auth/userpass/users/alice \
  password="123456" \
  policies="default"
```

Let's break down the path:

- `auth/` – This tells Vault we're interacting with an **auth method**.
- `userpass/` – This is the **path of auth method** we enabled earlier.
- `users/` – A sub-path used by the userpass plugin to manage users. It's fixed and you can't change it.
- `alice` – The **username** we're creating. You can choose whatever you want.

So essentially:

> Create a new user `alice` inside the `userpass` auth method and assign her a password and policy.



What Is the "default" Policy?

When you create a new user and assign `"default"` as the policy, you're giving them very limited capabilities. That's intentional.

The `default` policy typically allows users to:

- Authenticate (log in)
- Lookup their own token
- Renew and revoke their own token

But nothing else?????they **cannot** read or write secrets, manage engines, or do anything meaningful in your setup.

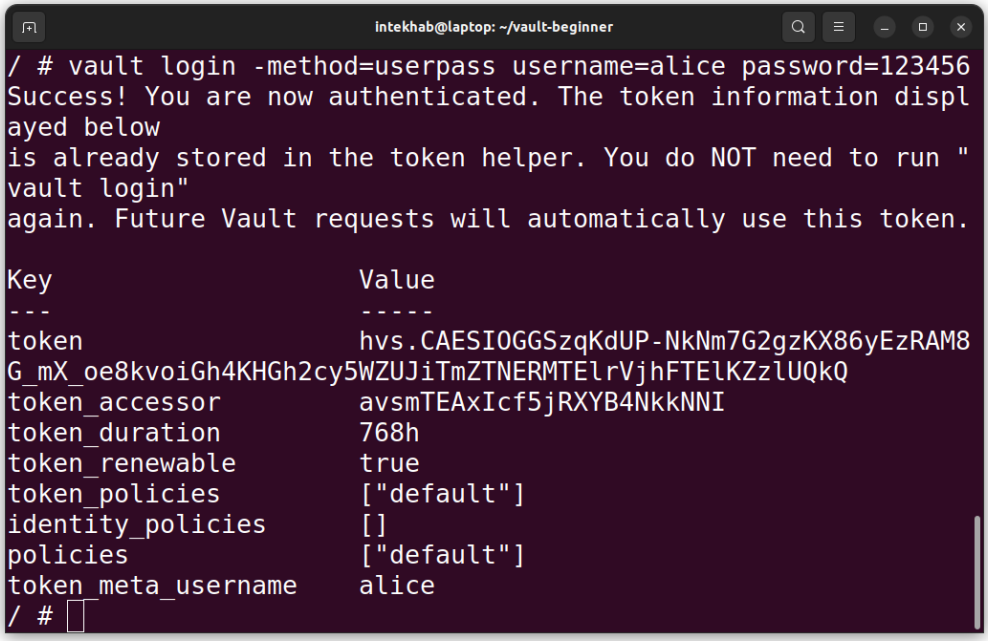
Vault plays it safe by design. Permissions must be explicitly granted.

Login as Alice

Now, let's switch roles. You're Alice. Login with the Alice's username and password.

```
vault login -method=userpass username=alice
password=123456
```

You'll get:



This token is Alice's key to the Vault. Scoped, limited, and short-lived.

Can Alice Access the Secret?

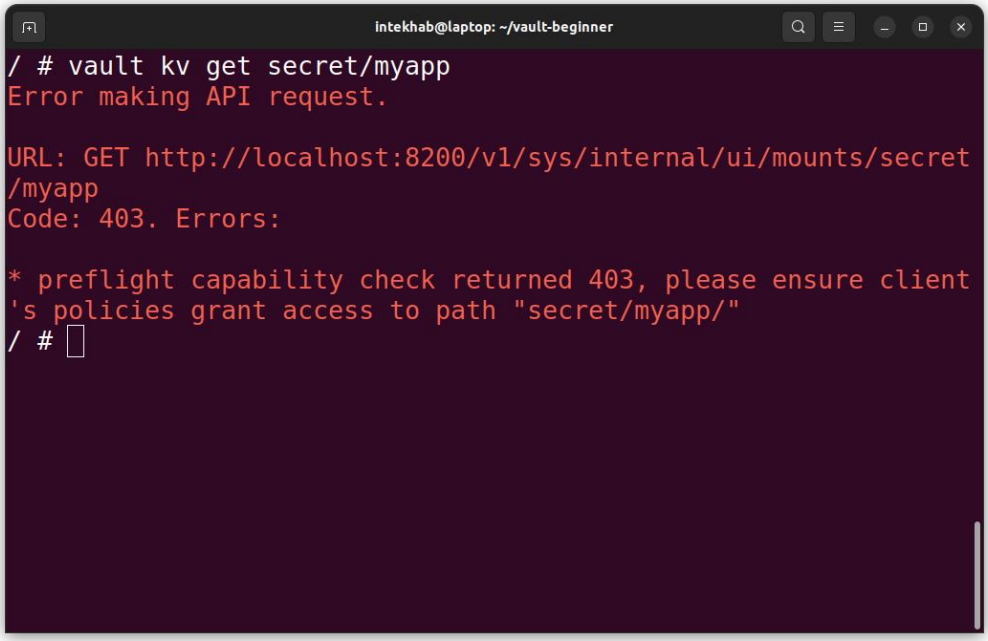
In the previous chapter, we stored a simple key-value secret at:

```
secret/myapp
```

Let's see if Alice can read it:

```
vault kv get secret/myapp
```

Result?



Boom. Permission denied.

Alice is authenticated but she's not authorised.

That's the Heart of Vault's Security Model

Vault always separates:

- **Authentication** – Who are you?
- **Authorization** – What are you allowed to do?

Just because Alice logged in doesn't mean she can read secrets. For that, she needs **access policies**. Which we'll build in the next chapter.

Remember

- Auth methods let users log in without using the root token.
- `userpass` is the easiest to try.

- `users/` is a special path inside the `userpass` method for managing user accounts.
- `default` policy is minimal, login only.
- Alice can log in but can't do anything useful yet (by design).

What's Next?

Now that we've got Alice in the system, let's give her access.

In the next chapters, we'll define **custom policies**. The Vault's way of saying:

> Yes, you may read this secret, but not that one.

Let's unlock the Vault carefully.

Chapter 9: Understanding Vault Policies

Vault doesn’t care who you are.

It cares what you can do, and that’s defined by **policies**.

You’ve already seen this in action: Alice couldn't access secrets until we gave her the right policies. But to truly master Vault, you must get comfortable writing and understanding these policies.

This chapter is your complete guide.

HCL - Hashicorp Configuration Language

Vault policies are written in **HCL**, a simple, human-readable language also used in Terraform. You can also use JSON, but HCL is more readable and preferred. Here’s a simple HCL policy:

```
path "kv/*" {
  capabilities = ["read", "list"]
}
```

You define **what path** a user can access and **what they can do** there.

Anatomy of a Policy Stanza

A single **policy stanza** looks like this:

```
path "kv/data/secret" {
  capabilities = ["create", "read", "update"]
}
```

Let's break that down:

- `path` – The Vault API path this rule applies to.
- `capabilities` – What the user is allowed to do at this path.

You can have multiple stanzas in one policy. Each one targets a specific path.

The Wildcards: `*` and `+`

These two symbols help match paths dynamically:

Symbol	Meaning
`*`	Wildcard for **one level** (no slash)
`+`	Wildcard for **multiple levels**, including slashes

Example:

```
path "kv/*" { }         # Matches: kv/foo, kv/bar (but
not kv/foo/bar)
path "kv/+/app" { }         # Matches: kv/foo/app,
kv/bar/app, kv/baz/app but not kv/foo/bank
```

Use `+` when you're dealing with unknown or deep hierarchies.

Capabilities - What Can You Do?

Vault understands these permissions:

Capability	Description
`create`	Write a new secret where none exists
`update`	Modify an existing secret
`read`	Read data (like secrets)
`delete`	Delete the data

Capability	Description
`list`	See what keys exist at a path
`sudo`	Full access, including overriding ACLs
`deny`	Explicitly block access, even if allowed elsewhere

You can combine them like this

```
capabilities = ["read", "list"]
```

How Policy Enforcement Works

When a request comes into Vault:

1. Vault checks **the identity** (token, user, etc.).
2. It looks at **all policies** attached to that identity.
3. It determines whether the action on that path is allowed.

> If any policy allows the action, it proceeds. Unless... An explicit `deny` exists, which **overrides all allows.**

Deny Always Wins

Vault uses **default-deny**; if a policy doesn't say something is allowed, it's denied. But if you explicitly write `deny`, it becomes unskippable.

```
path "kv/secret/*" {
  capabilities = ["read"]
}
path "kv/secret/*" {
  capabilities = ["deny"]
}
```

Even if a user has permission for `kv/secret/*`, the `deny` for `kv/secret/*` wins.

This is useful when you want to broadly allow access but block certain sensitive paths.

Deny in Any Policy Blocks Everything

Here's the key rule:

> If any policy attached to a user includes a `deny` for a path, access is denied, even if another policy allows it.

So if you assign `dev-policy` and `secure-policy` to a user, and `secure-policy` has a deny rule, it takes precedence.

Use `deny` carefully. It's a scalpel, not a hammer.

Example Policy: Reader with a Restriction

```
# Allow read/list on all kv paths
path "kv/+" {
  capabilities = ["read", "list"]
}

# Deny access to production secrets
path "kv/production/*" {
  capabilities = ["deny"]
}
```

This allows broad access, except to sensitive areas. Neat and clean.

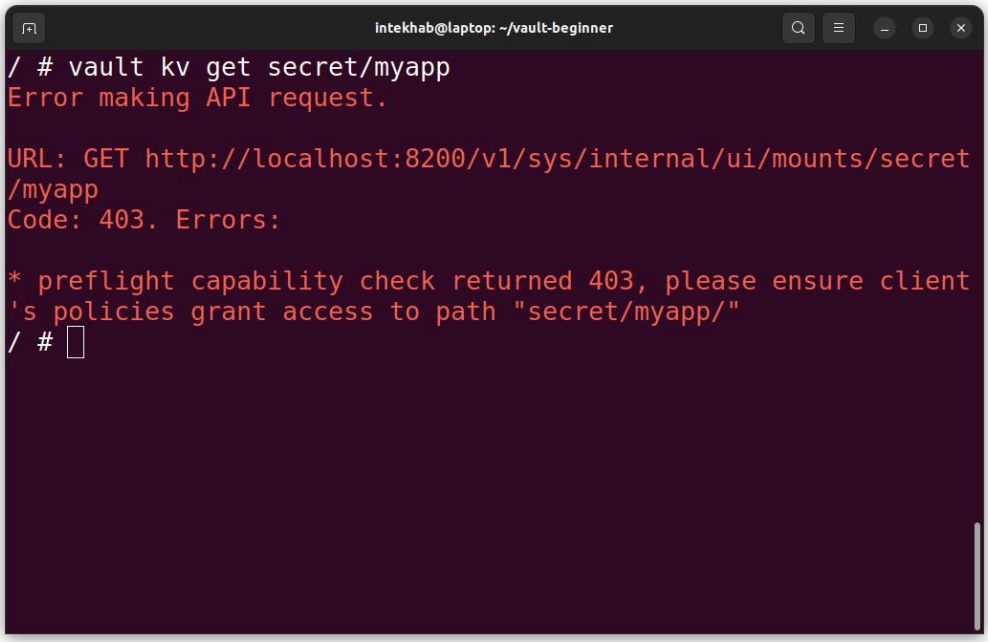
Summary

- Policies are written in **HCL**, with stanzas for each path.

- You define **what actions** users can perform using `capabilities`.
- Use `*` and `+` to target path patterns.
- `deny` trumps everything; even if another policy allows access.
- All policies assigned to a user are merged, and the most restrictive one wins.

Chapter 10: Policies in action

In the previous chapter, Alice logged in successfully. But when she tried to read the secret we stored? Failed.



Vault did precisely what it's supposed to do.

Every user or app must be explicitly granted access via **policies**, Vault's primary authorization mechanism.

Now it's time to fix that but **we can't do it as Alice**.

Important: Log Out as Alice and Log Back In as Root

Since Alice only has the `default` policy, she **can't**:

- Write or edit policies

- Assign policies to other users
- Manage Vault configuration

So first, we need to **switch back to the root token** or an admin-level token you used during initialization:

```
vault login
# Paste your root token when prompted
```

You should see something like:

```
Success! You are now authenticated.
token:      s.rootxxxxxxxx
policies:   [root]
```

Now we can proceed.

What is a Policy?

A **policy** is a set of rules that define:

- What paths in Vault a user or token can access
- What actions (read, write, list, delete, etc.) they can perform

You write policies in HCL or JSON format and assign them to users, tokens, or apps.

Step 1: Create the `kv1-reader` Policy

This policy gives **read-only** access to KV v1 secrets stored under the `secret/` path.

Create a file named `kv1-reader.hcl`:

```
# kv1-reader.hcl
```

```
path "secret/*" {
  capabilities = ["read", "list"]
}
```

To speed up the hands-on experience, we've already created the policy files for you and placed them inside the **`chapter-09`** folder. If you're using the Docker start command from Chapter 2, the **`vault-beginner`** folder is automatically mounted inside the Vault container at **`/home/vault/vault-beginner`**. From now on, before executing any commands mentioned in the upcoming chapters, make sure you first navigate to this directory.

Apply the policy to Vault:

```
vault policy write kv1-reader chapter-09/kv1-
reader.hcl
```

Success! Policy uploaded.

Step 2: Create the `kv2-reader` Policy

KV v2 paths are structured differently. You need access to `data/` for secrets and `metadata/` for listing.

Here's the `kv2-reader.hcl`:

```
# kv2-reader.hcl
path "kvv2/data/*" {
  capabilities = ["read"]
}

path "kvv2/metadata/*" {
  capabilities = ["list"]
}
```

Apply it:

```
vault policy write kv2-reader chapter-09/kv2-
reader.hcl
```

Done.

Step 3: Assign Both Policies to Alice

Now, let's update Alice's record to assign both policies (`kv1-reader` and `kv2-reader`):

```
vault write auth/userpass/users/alice \
  password="123456" \
  policies="kv1-reader,kv2-reader"
```

> Reminder: Always include the password during updates or it gets wiped out.

Step 4: Verify Policies Assigned to Alice

To confirm which policies are assigned:

```
vault read auth/userpass/users/alice
```

Look for Policies:

Looking good.

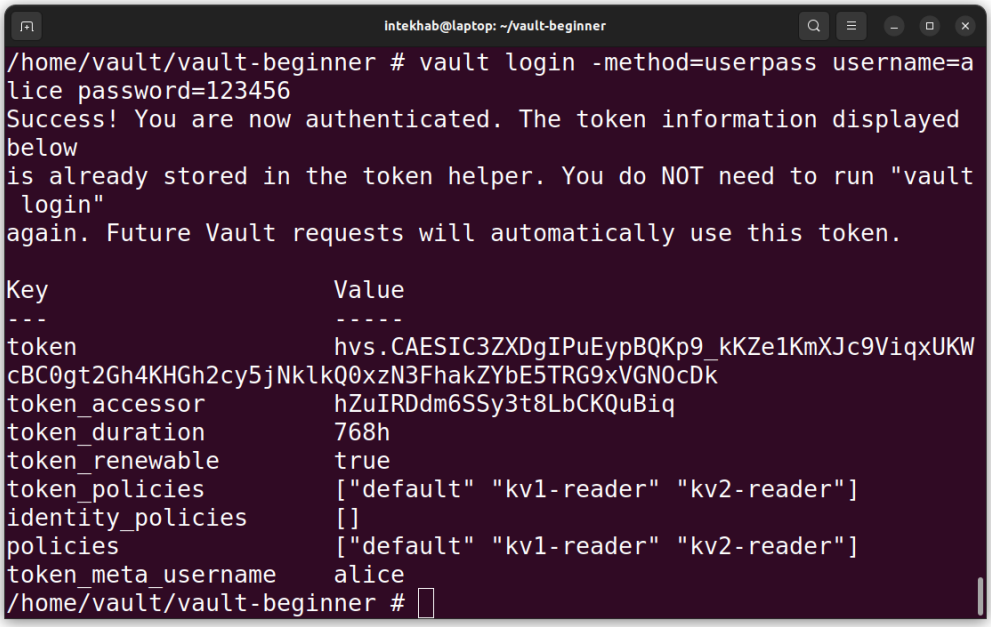
Step 6: Log Back In as Alice

Now we're done with the root user.

Let's log back in as Alice:

```
vault login -method=userpass username=alice
password=123456
```

You should now see. Look at the policies assigned to user. Now instead of just default policy, you will see 2 more policies attached to the user.



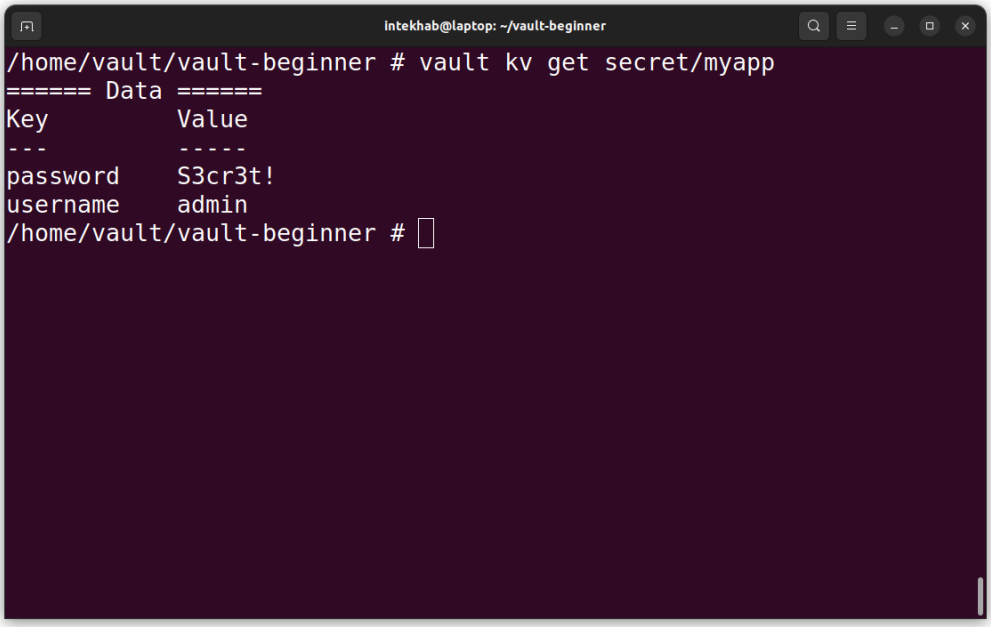
Step 7: Test Access as Alice

Let's try reading both types of secrets.

First, let's read Kv-v1 secret engine.

```
vault kv get secret/myapp
```

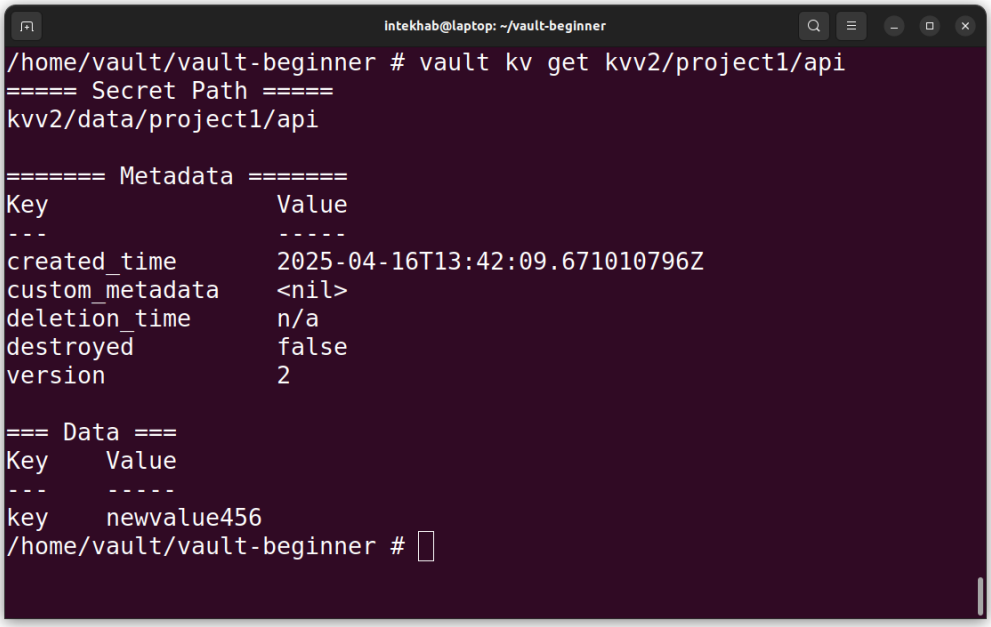
It should work and you would see the result like below.



Now Let’s try Kv-v2 secret engine. Execute below command.

```
vault kv get kvv2/project1/api
```

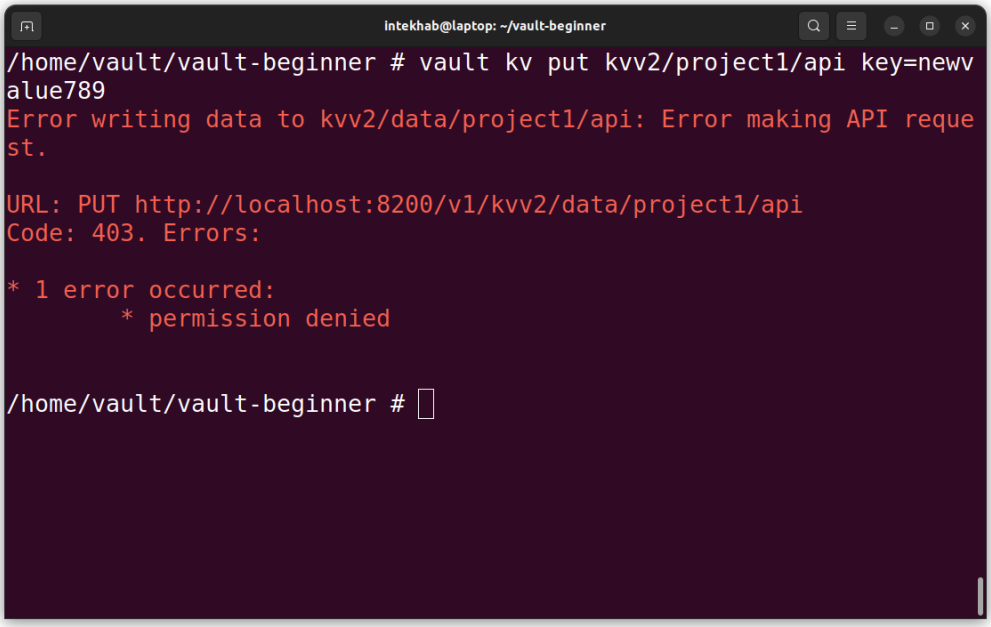
It should work and you would see the result like below.



Now, let's try to write something inside the kv-v2 secret engine. We will use the same command we have used earlier. Let's see what happens.

```
vault kv put kvv2/project1/api key=newvalue789
```

It should fail because we have given Alice just read-only access.



Summary

- Alice had no access initially because she only had the `default` policy.
- We **switched back to the root user** to create and assign new policies.
- Gave Alice access to both KV v1 and KV v2 secrets through separate policies.
- Saw how to list, inspect, and assign policies.
- Verified everything by logging in as Alice and testing access.

Chapter 11: Accessing Vault using browser-based UI

So far, we've been interacting with Vault exclusively using the command line. That’s powerful—but let’s be honest: sometimes, we all appreciate a nice visual interface. In this chapter, we’ll explore the **Vault UI**, which provides a point-and-click way to manage secrets, auth methods, and more.

Let’s talk about the different ways to access Vault and what the UI is (and isn’t) good for.

Vault provides three primary interfaces:

Interface	Description	Best Use Case
CLI	The command-line interface we've used so far.	Day-to-day admin tasks, scripting, automation.
HTTP API	The most complete way to interact with Vault.	Used by applications and integrations.
UI (Web Interface)	A browser-based dashboard.	Good for exploring, learning, and quick operations.

Important: The Vault UI is the least feature-rich interface. Some advanced features (like AppRole pull-based Secret ID) can only be done via CLI or API.

Enabling and Accessing the UI

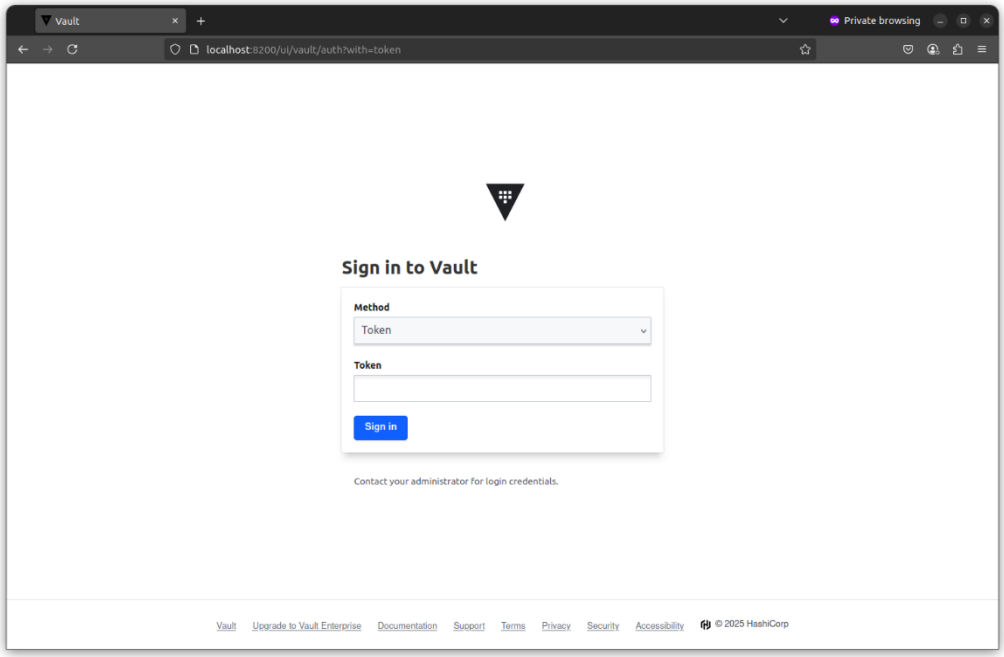
The Vault UI is **enabled by default** in most Vault images if the config has the following line. Have a look at your config file present inside the vault-data/config/vault-config.hcl folder

```
ui = true
```

In case you want to **disable the UI**, simply set it to false in your config file and restart the container using our same docker start command.

```
ui = false
```

That's it. Disabling the UI might be useful in high-security production environments.



Once your container is running with `ui = true`, you can access the UI by opening below URL in your browser.

```
http://localhost:8200
```

You can login at vault from UI by using the **root token** you received during `vault operator init`.

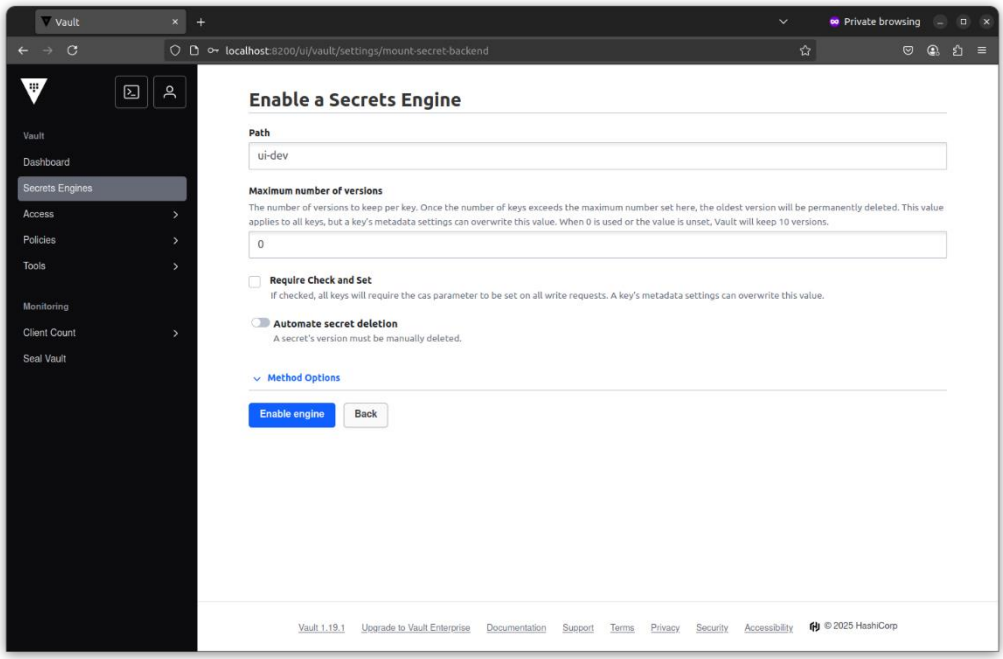
Hands-On – Let's use the UI to Manage a Secret Engine

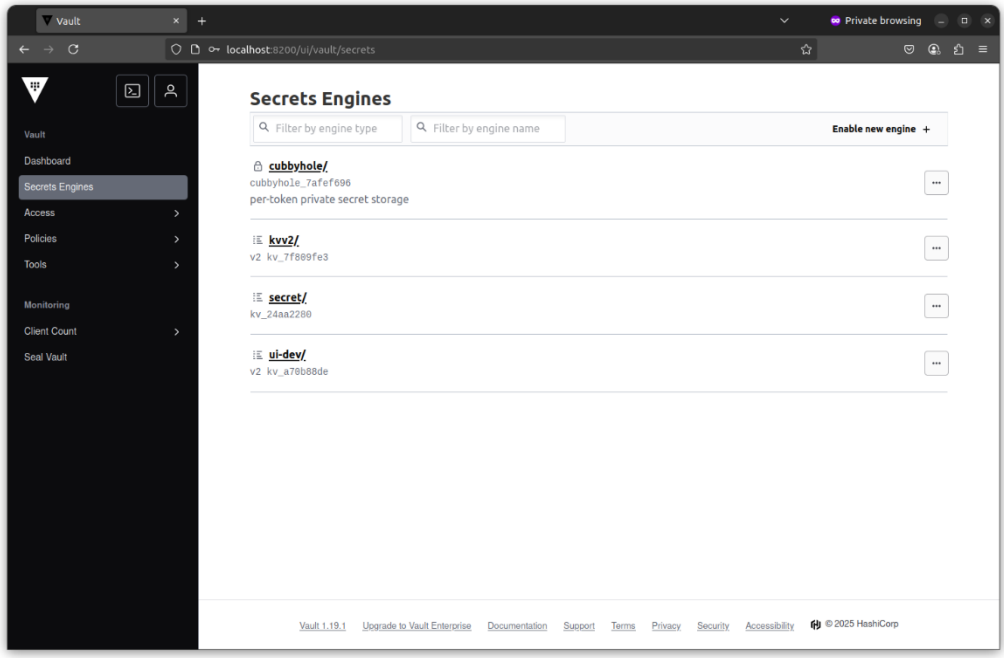
Let's now perform some real operations using the UI. You've already done these with the CLI—now let's repeat them visually.

1. Enable KV Secret Engine at `ui-dev`

1. Go to **Secrets Engines → Enable new engine**
2. Choose **KV**
3. Set **Path** to **`ui-dev`** and let all the value as is.
4. Click **Enable Engine**

Done. You've created a versioned secret engine at `ui-dev`.

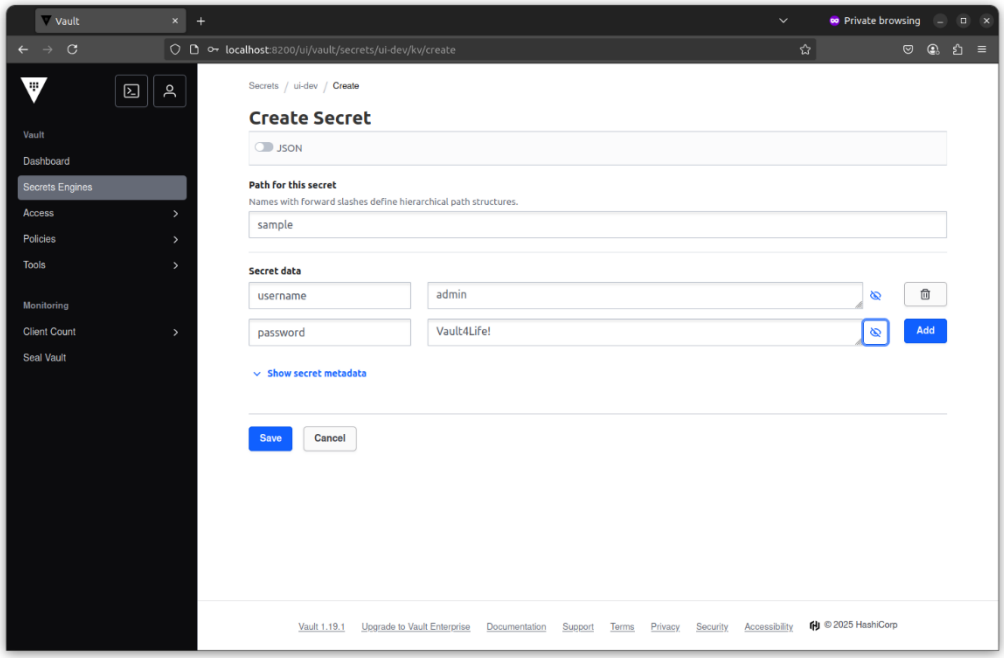

Vault
Private browsing
localhost:8200/ui/vault/settings/mount-secret-backend
Enable a Secrets Engine
Path
ui-dev
Maximum number of versions
The number of versions to keep per key. Once the number of keys exceeds the maximum number set here, the oldest version will be permanently deleted. This value applies to all keys, but a key's metadata settings can overwrite this value. When 0 is used or the value is unset, Vault will keep 10 versions.
0
Require Check and Set
If checked, all keys will require the cas parameter to be set on all write requests. A key's metadata settings can overwrite this value.
Automate secret deletion
A secret's version must be manually deleted.
Method Options
Enable engine
Back
Vault
Dashboard
Secrets Engines
Access
Policies
Tools
Monitoring
Client Count
Seal Vault
Vault 1.19.1
Upgrade to Vault Enterprise
Documentation
Support
Terms
Privacy
Security
Accessibility
© 2025 HashiCorp


Vault
Private browsing
localhost:8200/ui/vault/secrets
Secrets Engines
Filter by engine type
Filter by engine name
Enable new engine +
cubbyhole/
cubbyhole_7afef696
per-token private secret storage
kvv2/
v2 kv_7f889fe3
secret/
kv_24aa2280
ui-dev/
v2 kv_a70b88de
Vault
Dashboard
Secrets Engines
Access
Policies
Tools
Monitoring
Client Count
Seal Vault
Vault 1.19.1
Upgrade to Vault Enterprise
Documentation
Support
Terms
Privacy
Security
Accessibility
© 2025 HashiCorp

2. Put a New Value

1. Click into `ui-dev`
2. Click **Create secret**
3. Set the "**Path for this secret**" to `sample`
4. Under **Secret data**, add **key**: `username`, **value**: `admin`
5. Add another key: `password`, value: `Vault4Life!`
6. Click **Save**

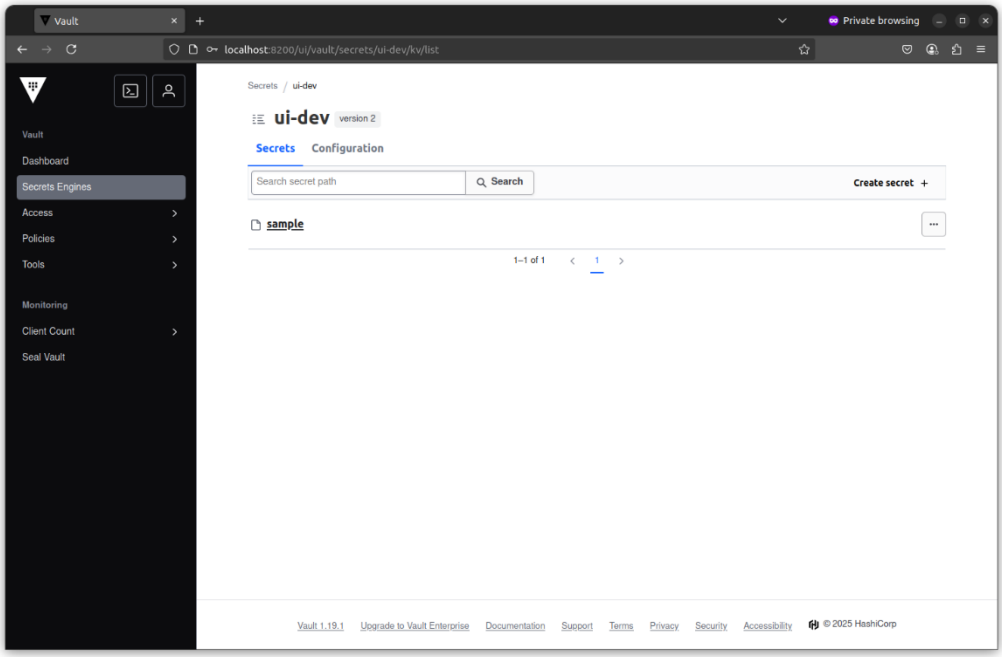
You've just written data to `ui-dev/sample`.



3. List All Values

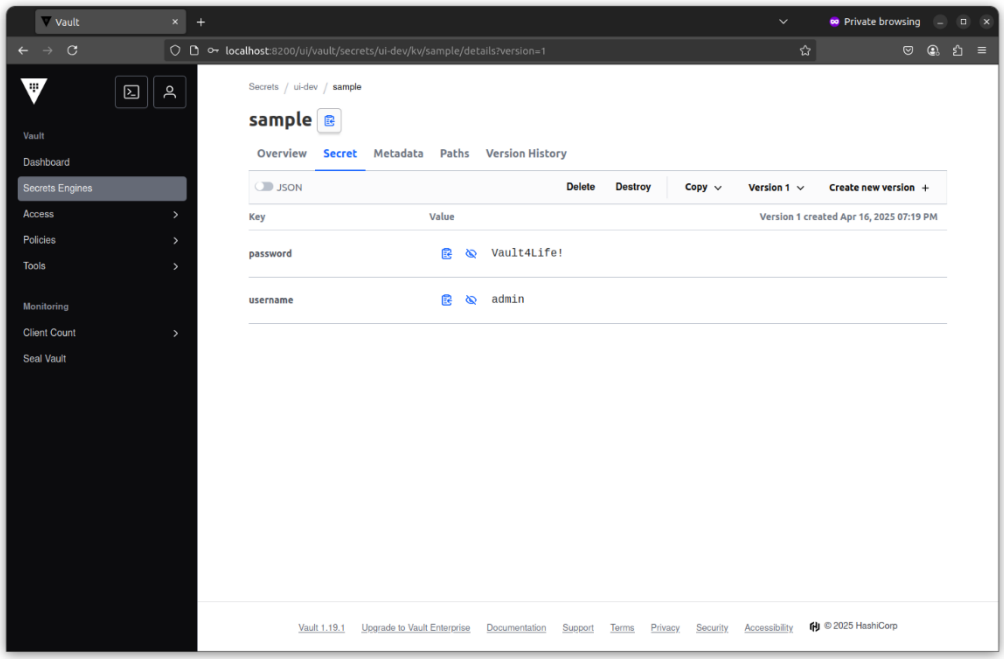
1. From the "**Secret Engines**" home page, click on "**ui-dev**"
2. You'll see all paths inside this engine

3. Each listed path is a secret entry



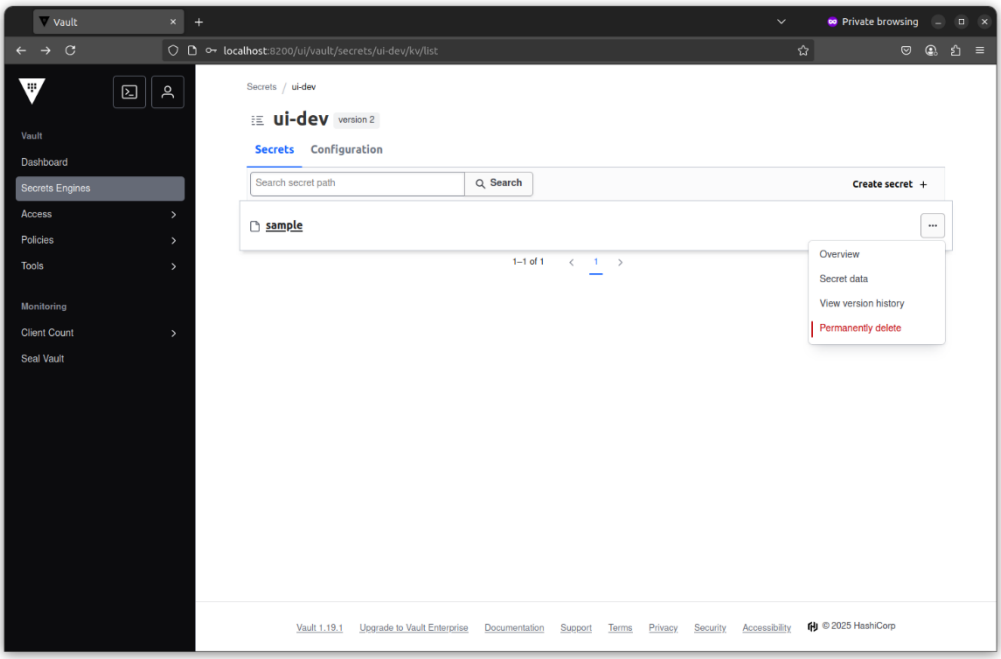
4. Read the Value

1. Now click on the "**Secret**" tab, you'll see the key-value pairs
2. You can copy the values from here for use elsewhere



5. Delete a Value

1. Now go back to the list page
2. Choose the three-dot menu (⋮) aginst the “**Sample**” avaliable on the right handside.
3. Click on **“Permanently delete”**

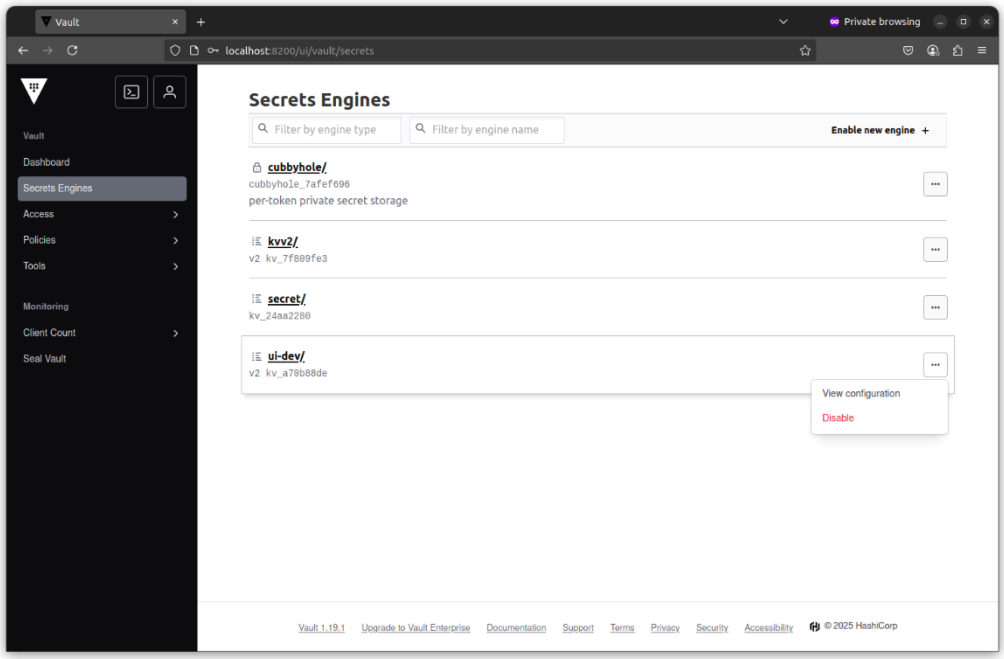


6. Disable the Secret Engine

Once you're done experimenting:

1. Go to **Secrets**
2. Find `ui-dev` in the list
3. Click the three-dot menu (⋮)
4. Click **Disable**

This will completely remove the secret engine and all the secrets under it.



Recap

- Vault offers **CLI**, **HTTP API**, and **UI** interfaces.
- The **API** is the most powerful and flexible.
- The **UI** is useful for quick access and exploration but doesn't support every feature.
- You can **disable the UI** in production for tighter security.
- We practised enabling and managing secrets visually using the **KV v2** engine.

Even though the UI is convenient, remember that anything done through the UI can also be done via API calls. This is important when scaling up or automating Vault in real systems.

Chapter 12: Understanding Dynamic Secrets

Imagine you're working in a growing team. You've got developers, DevOps folks, and maybe even a few interns. They all need access to sensitive systems, databases, cloud accounts, message queues, internal APIs...

So you do what most teams do:

- Create a shared DB user.
- Give out long-lived cloud keys.
- Hope everyone remembers to rotate them.
- Panic when someone leaves the company.

Now, enter **Vault's dynamic secrets,** your new best friend.

What Are Dynamic Secrets?

Dynamic secrets are **ephemeral credentials** that Vault generates **on-demand**.

Instead of creating a user manually and storing their password in Vault (a static secret), Vault talks to your system (like a database or cloud provider), creates a brand-new user with a random password, hands it over to the requester, and then **destroys it** after some time.

These secrets:

- Are **unique per request**

- Have a **short lifespan**
- Are **automatically revoked**
- Leave **zero traces** once expired

It's like printing a backstage pass that self-destructs after the show.

What Systems Can Use Dynamic Secrets?

Vault supports dynamic secret generation for:

Secret Engine	Use Case
Databases	Generate DB users for PostgreSQL, MySQL, MSSQL, etc.
Cloud Platforms	AWS, Azure, GCP – create short-lived API keys or IAM roles
SSH	One-time SSH access for remote machines
PKI	Create short-lived TLS certificates for apps
TOTP	Time-based OTP generation
RabbitMQ / MongoDB / Cassandra	Temporary app credentials

More integrations are constantly added. If your system can create users or rotate credentials, Vault can probably automate it.

Why Use Dynamic Secrets?

Here's what makes dynamic secrets a game-changer:

1. Time-Limited Access

No more infinite credentials lying around. You request access, use it, and Vault automatically revokes it after a set TTL (Time-To-Live).

2. Zero Sharing

Everyone gets their own secret. No more shared logins floating around in Slack or email.

3. Audit-Ready

Every issued secret has a traceable path: who requested it, when, and for what.

4. Kill Switch

If something smells fishy, you can revoke the secret instantly, like shutting off a keycard in a hotel.

5. Perfect for Automation

CI pipelines, temporary microservices, even short-lived development environments. Vault can create and destroy credentials on the fly.

Real-World Scenarios

Example 1: Dev Access to Databases

A developer needs read access to the production PostgreSQL database for 20 minutes.

- Vault creates a user like `v-token-readonly-12X9` with a random password.
- Dev connects using it.

- After 20 minutes, Vault deletes that user from PostgreSQL.
- Nothing to clean up. No password to rotate.

Example 2: CI/CD Deployment to AWS

Your deployment pipeline needs to upload files to S3.

- Instead of hardcoding an IAM key in GitHub Actions, your pipeline logs in to Vault.
- Vault generates temporary AWS keys with scoped permissions.
- The job runs. Vault revokes the key after 15 minutes.
- Your AWS account stays clean and locked down.

Example 3: Secure Onboarding

A new engineer joins the team.

- You give her Vault access with limited policies.
- She requests credentials for staging systems through Vault.
- No need to update .env files, no shared secrets, no accidental leaks.

The Payoff

With dynamic secrets:

- Your secrets stay fresh.
- Your blast radius shrinks.
- Your compliance posture improves.
- Your security team sleeps better.

It’s one of those features that makes you rethink how secrets should work.

Now that your mind is prepped, in the next chapter, we’ll **get hands-on**. We’ll connect Vault to a real system (like Mysql), configure the secret engine, create a dynamic role, and watch Vault generate credentials on the fly.

You’re going to love it.

Let’s go build it.

Chapter 13: Dynamic Secrets in Action

Now that we understand what dynamic secrets are and why they're game-changing, it's time to get hands-on.

We'll configure Vault to dynamically generate **MySQL credentials**. No manual provisioning, no password sharing. And we'll **watch it happen in real-time using phpMyAdmin**.

Let's dive in.

Step 1: Set Up Your Playground with Docker Compose

We'll start:

- Vault
- MySQL (with a root password and a test database)
- phpMyAdmin (to visually track user creation)

Create a file named `docker-compose.yml`:

> *To speed up the hands-on experience, we've already created the* ***`docker-compose.yml`*** *files for you and placed them inside the* ***`chapter-13`*** *folder. Make sure to go inside the chapter-13 folder before executing the* ***docker compose up -d*** *command.*

```
services:
  vault:
    image: hashicorp/vault:latest
    container_name: vault
    ports:
      - "8200:8200"
    environment:
      VAULT_DEV_LISTEN_ADDRESS: "0.0.0.0:8200"
    volumes:
      - ../vault-data/config:/vault/config
      - ../vault-data/file:/vault/file
      - ../vault-data/logs:/vault/logs
      - ../:/home/vault/vault-beginner
    cap_add:
      - IPC_LOCK
    command: "server"

  mysql:
    image: mysql:8.0
    container_name: mysql
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: vaultdb
    ports:
      - "3306:3306"

  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    container_name: phpmyadmin
    environment:
      PMA_HOST: mysql
      MYSQL_ROOT_PASSWORD: rootpass
    ports:
      - "8080:80"
```

Step 2: Stop the old vault-dev container

If the container from the previous chapter is still running, you'll need to stop it before starting a new one.

Run:

```
docker stop vault-dev
```

This ensures there's no port conflict and a clean start for your new setup.

Step 3: Start the Stack

Run:

```
docker compose up -d
```

This command will start three containers: HashiCorp Vault, MySQL Database, and phpMyAdmin. Once they're up and running, you can access the Vault UI and phpMyAdmin via the following URLs in your browser:

- **Vault UI** → http://localhost:8200
- **phpMyAdmin** → http://localhost:8080
  (login as `root` / `rootpass`)

Step 4: Attach to the Vault Container

To interact with Vault CLI inside the container, run below command.

```
docker exec -it vault-dev /bin/sh
```

Now you're inside the Vault container.

Step 5: Export the Vault Address

Before using Vault CLI, we need to tell it **where** Vault is running.

```
export VAULT_ADDR=http://localhost:8200
```

This command sets the Vault server address as an environment variable. The CLI uses this to send all your requests to Vault.

Step 6: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

> ***Note:*** *Use the unseal keys you saved during initialization. If those keys are lost, you'll need to reinitialize Vault from scratch.*

Step 7: Login to Vault

Login as root

```
vault login <your-root-token>
```

(Replace `<your-root-token>` with your actual root token you have received when you performed the vault operator init.)

Step 8: Enable the Database Secrets Engine

Run:

```
vault secrets enable database
```

This enables the `database/` path, which Vault uses to manage connections and dynamic secrets for databases.

Step 9: Configure the MySQL Plugin

Here's how we teach Vault to talk to our MySQL database:

```
vault write database/config/my-mysql-database \
  plugin_name=mysql-database-plugin \

connection_url="{{username}}:{{password}}@tcp(mysql:3
306)/" \
  allowed_roles="readonly-role" \
  username="root" \
  password="rootpass"
```

Let's break it down:

🔹 **my-mysql-database**

This is just a logical name for your connection config. You'll refer to this again later when creating roles (using `db_name=my-mysql-database`). Think of it like a reference alias.

🔷 **`plugin_name`**

This tells Vault which plugin to use. For MySQL, it's always `mysql-database-plugin`.

🔷 **`connection_url`**

Vault uses this format to log into your MySQL server. The placeholders `{{username}}` and `{{password}}` will be replaced by the values below?????the root user and password.

The `mysql:3306` is the container name and port inside Docker.

🔷 **`allowed_roles`**

This field whitelists the roles that are allowed to use this database configuration.
Only these roles (like `readonly-role`) will be permitted to generate credentials for this DB. This is a critical security control.

Step 10: Define the Dynamic Credentials Role

Now, we create a **role** that defines how Vault should create MySQL users.

```
vault write database/roles/readonly-role \
  db_name=my-mysql-database \
  creation_statements="CREATE USER '{{name}}'@'%'
IDENTIFIED BY '{{password}}'; GRANT SELECT ON
vaultdb.* TO '{{name}}'@'%';" \
  default_ttl="1m" \
  max_ttl="5m"
```

Let's understand this:

🔷 **`db_name=my-mysql-database`**

This links the role to the database config we created earlier (with the name `my-mysql-database`).

🔷 **`creation_statements`**

This is the real magic?????it tells Vault **how** to create a user in SQL.

```
CREATE USER '{{name}}'@'%' IDENTIFIED BY
'{{password}}';
GRANT SELECT ON vaultdb.* TO '{{name}}'@'%';
```

- `{{name}}` and `{{password}}` are placeholders that Vault replaces when it generates a user.
- This user gets **read-only** access to everything in the `vaultdb` database.
- The user is valid from **any host** (because of `@'%'`).

🔷 **`default_ttl` and `max_ttl`**

These define how long the user exists:

- Default time-to-live: 1 minute
- Max time-to-live: 5 minutes

Vault will automatically delete the user after the TTL expires unless the lease is renewed.

Step 11: Generate Dynamic Credentials

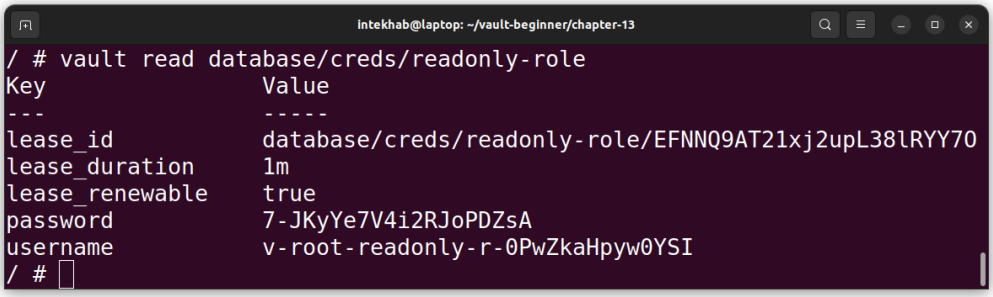
Now we ask Vault to give us a new MySQL credential:

```
vault read database/creds/readonly-role
```

This triggers Vault to:

1. Generate a new MySQL user.
2. Insert it into MySQL.
3. Return the credentials to you.

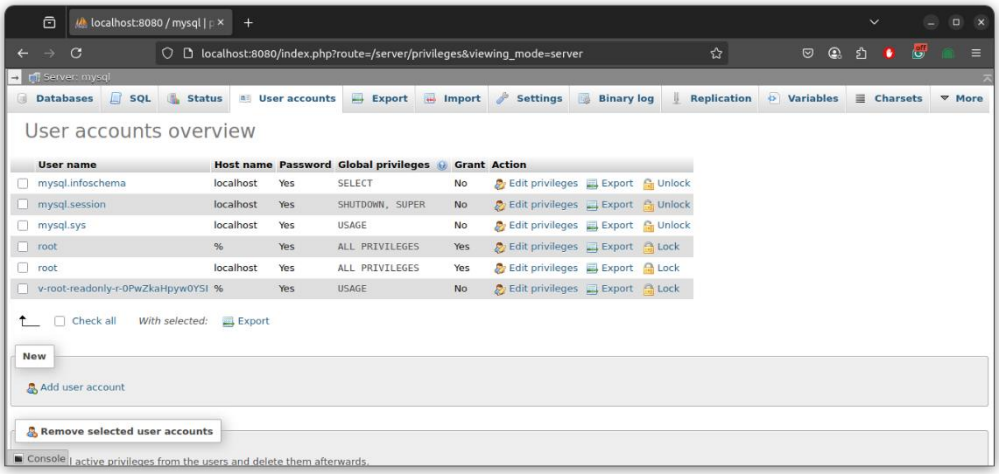
Example output:



Let's explain what just happened:

- `vault read database/creds/readonly-role`: This asks Vault to create a dynamic secret for the `readonly-role`.
- Vault uses the role's `creation_statements` to make a new user.
- That user now exists **in MySQL** and can only read from `vaultdb`.

Open **phpMyAdmin**, go to the **Users** tab, and you'll see the new user created by Vault!



Step 12: Watch TTL Expiry in Real Time

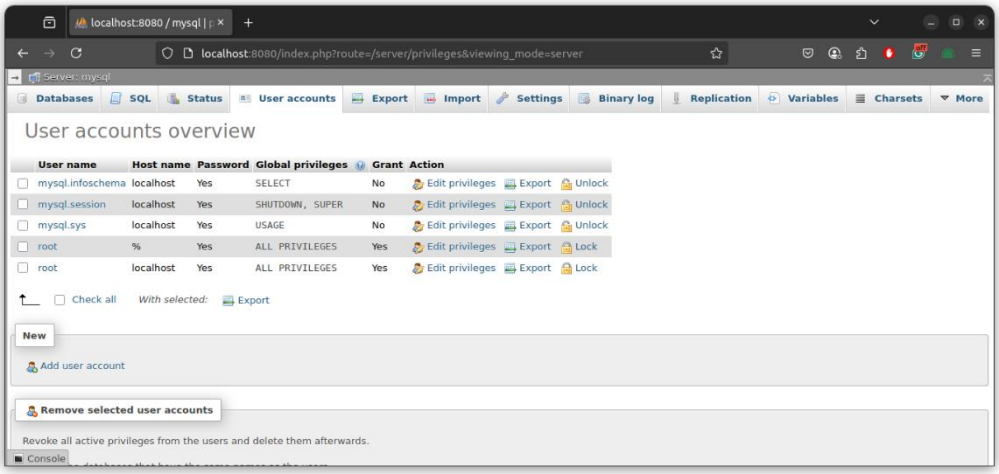
Wait 1 minute and refresh the users list in phpMyAdmin.
Poof. The user is gone.

That’s Vault cleaning up after itself?????just like it promised.

You can also revoke manually:

```
vault lease revoke <lease_id>
```

Replace `<lease_id>` with the one you got in the output above.



Recap

What we did:

- Created a local lab with Vault, MySQL, and phpMyAdmin.
- Configured Vault to connect to MySQL using secure root credentials.
- Defined a **role** that generates dynamic MySQL users.
- Saw those users appear and disappear inside phpMyAdmin?????like clockwork.

Next Up

In the next chapter, we'll:

- Tweak TTLs and permissions for dynamic users.
- Build more complex SQL creation statements.

- Use **Vault policies** to allow users (like Alice) to request dynamic DB credentials?????securely.

We're officially automating secure database access?????welcome to the next level.

Chapter 14: Fine-Tuning and Access Control on Dynamic Secrets

In the previous chapter, we watched Vault create MySQL users on the fly. Now, let's refine that setup and introduce secure delegation. We'll:

- Explore advanced TTL control
- Create multiple roles with different permissions
- Use policies to delegate access to non-root users (like Alice)

Understand TTL Options

When creating dynamic secrets roles, Vault supports two TTL settings:

```
default_ttl = "1m"
max_ttl     = "5m"
```

🔹 `default_ttl`

- This is how long the credential lasts **by default**.
- If the client doesn't explicitly renew it, Vault will revoke it after this period.

🔹 `max_ttl`

- The absolute upper limit for the secret's lifespan.

- Even with renewal, the secret cannot live beyond this.

Why this matters: If your app only needs short-lived access (e.g., quick backup scripts), this drastically reduces the blast radius in case of compromise.

Before you begin: Ensure your Vault environment from the previous chapter is up and running. If it isn't, please revisit and complete through Step 7 of that chapter. Once your Vault stack is fully initialized, unsealed and logged in as root, you can proceed with the steps below.

Step 1: Create a New Read-Write Role

Let's define a new role that gives **read and write** permissions on the MySQL database:

```
vault write database/roles/readwrite-role \
  db_name=my-mysql-database \
  creation_statements="CREATE USER '{{name}}'@'%'
IDENTIFIED BY '{{password}}'; GRANT SELECT, INSERT,
UPDATE ON vaultdb.* TO '{{name}}'@'%';" \
  default_ttl="2m" \
  max_ttl="10m"
```

Step 2: Update the Database Config to Allow New Read-Write Role

Let's re-write the database config and include our newly created readwrite-role in allowed_roles. Run below command

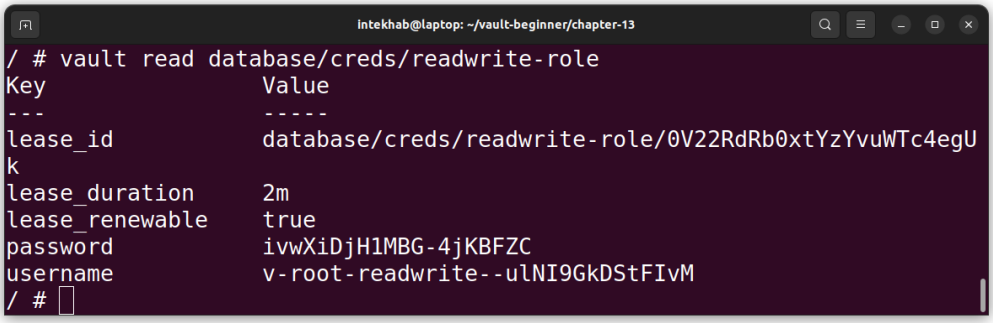
```
vault write database/config/my-mysql-database \
  plugin_name=mysql-database-plugin \

connection_url="{{username}}:{{password}}@tcp(mysql:3
306)/" \
  allowed_roles="readonly-role, readwrite-role" \
  username="root" \
  password="rootpass"
```

Step 3: Generate Dynamic Credentials

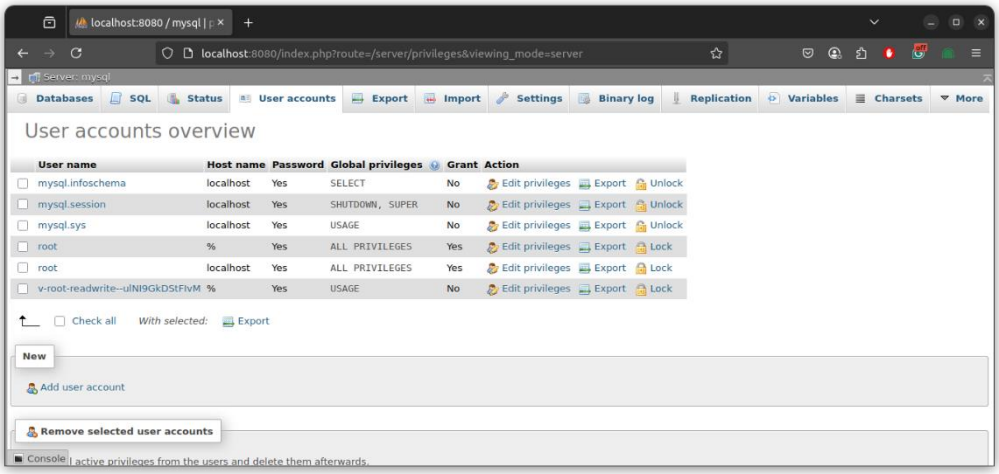
You can now generate a credential with:

```
vault read database/creds/readwrite-role
```

You will see the output something like below.



And if you check the PHPmyAdmin, you will see the same cred present in MySQL database users liet.



Step 4: Secure Access with Policies

Let's say you want Alice to be able to generate only **read-only MySQL credentials**.

Let's write a new policy for Alice. File name readonly-mysql-policy.hcl

```
# readonly-mysql-policy.hcl
path "database/creds/readonly-role" {
  capabilities = ["read"]
}
```

Create policy using the above policy file.

```
vault policy write readonly-mysql chapter-
14/readonly-mysql-policy.hcl
```

Attach the policy to Alice:

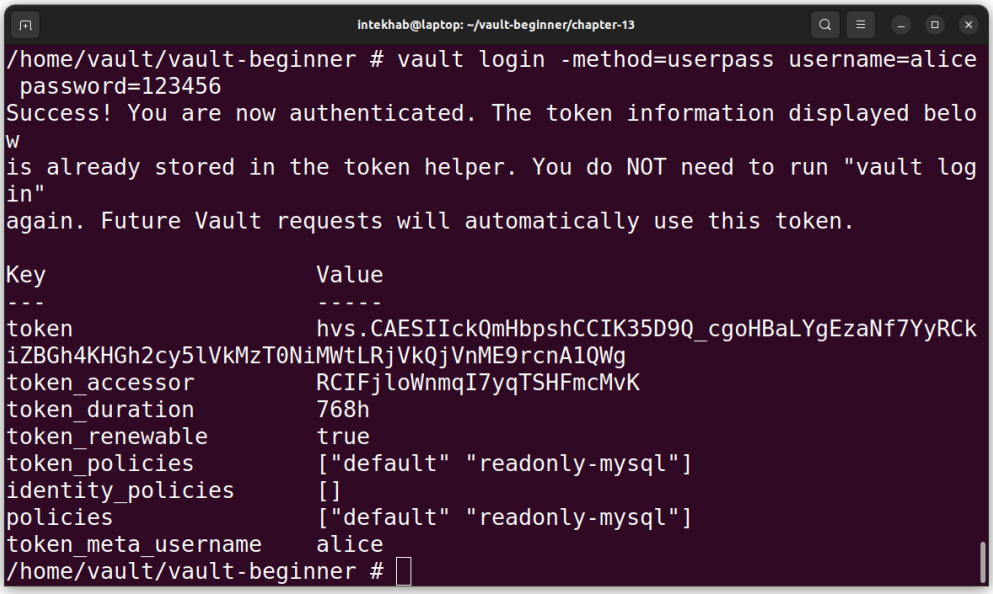
```
vault write auth/userpass/users/alice \
  password="123456" \
  policies="default,readonly-mysql"
```

Step 5: Validate Permissions

Log in as Alice:

```
vault login -method=userpass username=alice
password=123456
```

You will see the “readonly-mysql” policy is now attached to Alice.



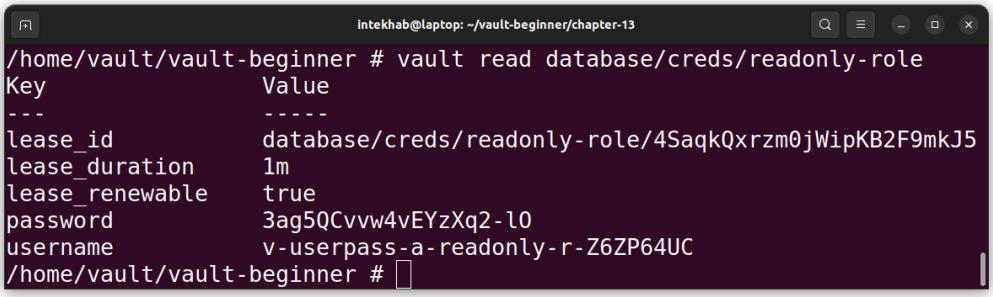
Step 6: Generate a MySQL User for Alice with Read-Only Access

Now let’s generate dynamic MySQL credentials for **Alice** using the `readonly-role`, which is configured to grant read-only access.

Run the following command:

```
vault read database/creds/readonly-role
```

You should see Vault return a newly created MySQL username and password, along with the lease duration—this account is tailored specifically for Alice with read-only privileges.

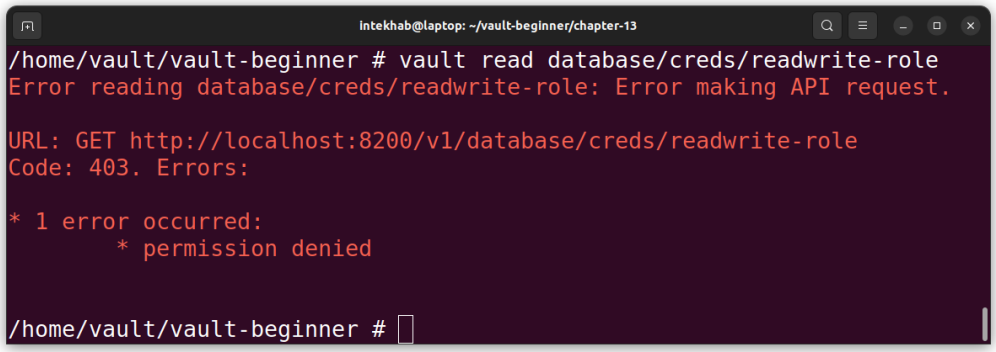


Step 7: Attempt to Generate a MySQL User for Alice with Read-Write Access

Next, try generating dynamic MySQL credentials for Alice using the `readwrite-role`, which is configured to grant read-write access:

```
vault read database/creds/readwrite-role
```

Vault will return a **permission denied** error—exactly as expected.



Bingo! This confirms that Vault's fine-grained access control is working. Alice doesn't have access to this role, so Vault blocks the request.

Wrap Up

You've now:

- Created roles with different DB access levels
- Controlled access via Vault policies
- Delegated MySQL access to a non-root user (Alice)

This is the Vault way:

- Least privilege
- Time-bound access
- Clear delegation without sharing root credentials

Coming up next: we'll explore **leases, renewals, and revocations** in dynamic secrets. How long can access live, and who decides when it dies?

Chapter 15: Leasing, TTL, and Vault's Secret Lifecycle

Now that you've seen Vault dynamically generate MySQL credentials for you (magic, right?), you might be wondering:

- How long do those credentials last?
- What happens when they expire?
- Can I revoke them early?
- Can I extend them?
- Can I restrict how many times they're used?

This chapter dives into Vault's **leasing mechanism**?????the core engine that governs how long secrets live, how they're revoked, and how they behave after being handed out.

What Is a Lease?

When Vault generates a dynamic secret (like our MySQL credentials), it also generates a **lease**. A lease is like a signed contract between you and Vault:

> "Here's your secret. You're allowed to use it for X seconds or Y number of times. After that, it's gone."

Every lease has:

- A `lease_id`
- A `lease_duration` (TTL)

- A `renewable` flag
- Optionally, a `num_uses` limit

Example:

```
lease_id:       database/creds/readonly-
role/abc123456
lease_duration: 1h
renewable:      true
```

TTL (Time to Live)

TTL determines how long the secret remains valid. After the TTL, Vault automatically revokes the secret.

TTL can be configured:

- On the **secret engine**
- On the **role**
- At the **Vault server level**

Example during role creation:

```
vault write database/roles/readonly-role \
    db_name=my-mysql-database \
    creation_statements="..." \
    default_ttl="1h" \
    max_ttl="24h"
```

- `default_ttl`: Applies to leases unless manually overridden.
- `max_ttl`: The maximum TTL even if a lease is renewed multiple times.

3. `num_uses`: Expire After X Uses

Besides TTL, Vault allows you to set a `num_uses` on a role?????which limits how many times a generated secret can be used, regardless of the TTL.

Once the credential is used the specified number of times, Vault **automatically revokes** the lease.

This is incredibly useful for:

- One-time login credentials
- Scripts or apps that only need brief access
- Preventing credential reuse in CI/CD pipelines

Here's how to set it correctly **during role creation**:

```
vault write database/roles/readonly-role \
    db_name=my-mysql-database \
    creation_statements="..." \
    default_ttl="1h" \
    max_ttl="24h"
    num_uses=1
```

`num_uses` : Every time a secret is issued using this role, that username/password will expire after **one use**?????no matter how much time is left on the TTL.

You can verify this behavior by issuing credentials:

```
vault read database/creds/readonly-role
```

Try using the username and password once?????it'll work.
Try again? It's already revoked.

4. Viewing Active Leases

To list leases for a role:

```
vault list sys/leases/lookup/database/creds/readonly-role
```

To look up lease details:

```
vault lease lookup <lease_id>
```

5. Renewing a Lease

If the lease is renewable and TTL hasn't hit `max_ttl`, you can extend the lease:

```
vault lease renew <lease_id>
```

This is especially useful for long-running applications or sessions that don't want to re-authenticate.

6. Revoking a Lease

Need to shut off access early? Use:

```
vault lease revoke <lease_id>
```

This invalidates the associated secret and revokes access on the spot.

You can also revoke everything under a path:

```
vault lease revoke -prefix database/creds/readonly-role
```

7. Lease vs Static Secrets

Let’s make the distinction clear:

Feature	Static Secret (e.g., KV)	Dynamic Secret (e.g., DB)
Manually managed	Yes	No
Automatically expired	No	Yes
TTL-based	No	Yes
num_uses support	No	Yes
Auto-revocation	No	Yes

8. Why Leases Matter

This leasing model gives you control, auditability, and peace of mind. Imagine:

- An app gets hacked. Revoke the lease = instant kill switch.
- You want a password to expire after one use. `num_uses = 1`.
- You rotate secrets every hour. TTL handles it for you.

Vault isn’t just storing secrets. It’s managing their lifecycle securely, on your behalf.

Chapter 16: Dynamic Secrets with PostgreSQL

Introduction

We've seen how Vault dynamically creates database credentials for MySQL. Now it's time to explore the same for PostgreSQL.

In this chapter, we'll:

- Spin up Vault, PostgreSQL, and Adminer using Docker Compose.
- Configure Vault to generate PostgreSQL users on the fly.
- Understand the configuration options like `creation_statements`, `allowed_roles`, and templating variables like `{{name}}`.
- Test the generated secrets with Adminer.

Step 0: Clean Up Previous Setup

Before we begin, let's make sure no conflicting containers are running from our previous exercises.

Run:

```
docker compose down
```

This will stop and remove all existing containers. We're starting fresh.

To speed up the hands-on experience, we've already created the **`docker-compose.yml`** files for you and placed them inside the **`chapter-16`** folder. Make sure to go inside the **chapter-16** folder before executing the **docker compose up -d** command.

Step 1: Docker Compose Playground

Here's our new `docker-compose.yml` to spin up Vault, PostgreSQL, and Adminer (our DB web UI):

```
services:
  vault-dev:
    image: hashicorp/vault:latest
    container_name: vault-dev
    ports:
      - "8200:8200"
    environment:
      VAULT_DEV_LISTEN_ADDRESS: "0.0.0.0:8200"
    volumes:
      - ../vault-data/config:/vault/config
      - ../vault-data/file:/vault/file
      - ../vault-data/logs:/vault/logs
      - ../:/home/vault/vault-beginner
    cap_add:
      - IPC_LOCK
    command: "server"
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: rootpassword
      POSTGRES_DB: vaultdb
    ports:
      - "5432:5432"
  adminer:
    image: adminer
    container_name: adminer
```

```
    ports:
      - "8080:8080"
```

What's happening here?

- **Vault**: Starts our vault server.
- **Postgres**: Starts with:
    - Username: `root`
    - Password: `rootpassword`
    - Database: `vaultdb`
- **Adminer**: Lightweight web UI for interacting with the database.

Start everything:

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

Step 2: Attach to the Vault Container

To interact with Vault CLI inside the container, run below command.

```
docker exec -it vault-dev /bin/sh
```

Now you're inside the Vault container.

Step 3: Export the Vault Address

Before using Vault CLI, we need to tell it **where** Vault is running.

```
export VAULT_ADDR=http://localhost:8200
```

This command sets the Vault server address as an environment variable. The CLI uses this to send all your requests to Vault.

Step 4: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

> **Note:** Use the unseal keys you saved during initialization. If those keys are lost, you'll need to reinitialize Vault from scratch.

Step 5: Login to Vault

Login as root

```
vault login <your-root-token>
```

(Replace `<your-root-token>` with your actual root token you have received when you performed the vault operator init.)

Step 6: Enable Database Secrets Engine on a Custom Path

This time, we'll enable the **database secrets engine on a custom path** named **`pgdb`**.

```
vault secrets enable -path=pgdb database
```

Why use a custom path?

By default, Vault enables secrets engines at paths like `database/`, `kv/`, etc. But using a custom path (`pgdb` in this case) gives you flexibility:

- You can manage multiple databases (e.g., MySQL, PostgreSQL) side by side.
- You avoid name clashes if different roles or apps use different engines.
- It improves clarity and structure in your Vault layout.

> All of our subsequent Vault commands in this chapter will use the `pgdb/` path to interact with PostgreSQL.

Step 4: Configure Vault to Connect to PostgreSQL

Vault will need root credentials to manage PostgreSQL roles. Instead of hardcoding them, we'll use Vault templating to inject them via environment variables:

```
vault write pgdb/config/my-postgres-db \
  plugin_name=postgresql-database-plugin \
```

```
  allowed_roles="readonly-role" \

connection_url="postgresql://{{username}}:{{password}
}@postgres:5432/vaultdb?sslmode=disable" \
  username="root" \
  password="rootpassword"
```

Breakdown:

- `pgdb/config/my-postgres-db`: Path where this config is saved.
- `plugin_name`: Tells Vault what type of DB plugin to use.
- `allowed_roles`: Only these Vault roles are allowed to use this DB config.
- `connection_url`: Vault injects `{{username}}` and `{{password}}` from below.
- `username` and `password`: Used to authenticate with the DB (as root).

This setup gives Vault full control over the database.

Step 5: Create a Readonly Role

```
vault write pgdb/roles/readonly-role \
  db_name=my-postgres-db \
  creation_statements="CREATE ROLE \"{{name}}\" WITH
LOGIN PASSWORD '{{password}}' VALID UNTIL
'{{expiration}}'; GRANT SELECT ON ALL TABLES IN
SCHEMA public TO \"{{name}}\";" \
  default_ttl="1h" \
  max_ttl="24h" \
  num_uses=5
```

What's happening here?

- `pgdb/roles/readonly-role`: Our dynamic role path.
- `db_name`: Links to the `my-postgres-db` config we wrote earlier.
- `creation_statements`: SQL used to create the dynamic user.

Let's dissect the placeholders:

- `{{name}}`: Vault-generated username.
- `{{password}}`: Vault-generated password.
- `{{expiration}}`: Time when the credential should expire.

So every time this role is used, Vault will:

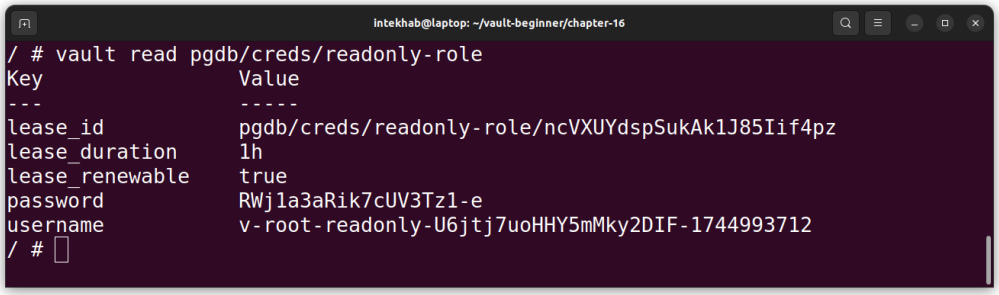
- Generate a unique username/password.
- Grant **read-only (SELECT)** access to the public schema.
- Set TTL and number of uses to expire/lock the credentials.

Step 6: Generate Dynamic Credentials

Run:

```
vault read pgdb/creds/readonly-role
```

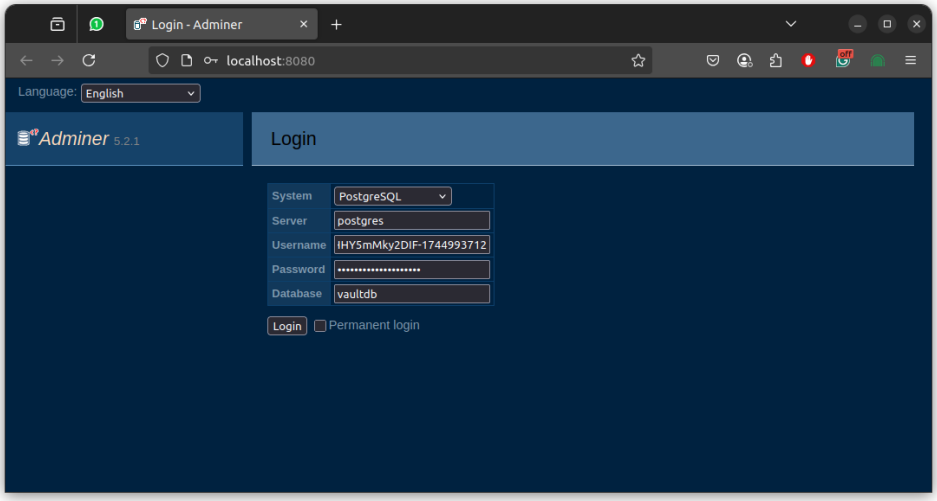
You will see output like below.



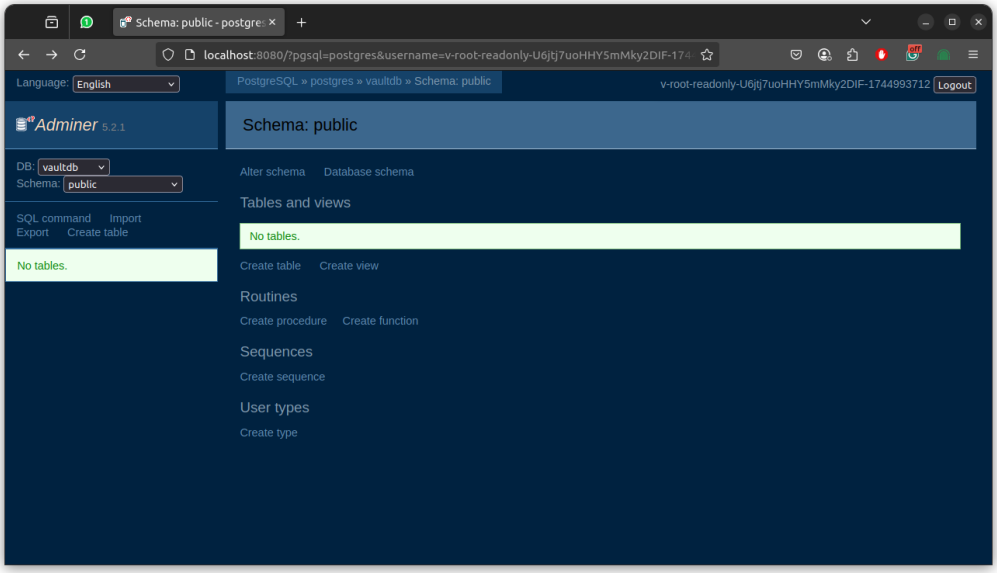
This credential is valid for 1 hour or **5 uses**—whichever comes first.

Step 7: Use It with Adminer

1. Go to http://localhost:8080
2. Fill in:
    - **System**: PostgreSQL
    - **Server**: postgres
    - **Username** and **Password**: From the Vault response
    - **Database**: `vaultdb`



3. Hit Login. You should land in the DB with read-only rights.

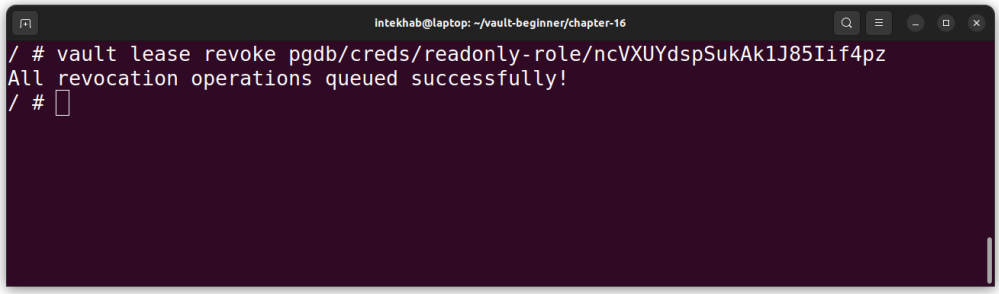


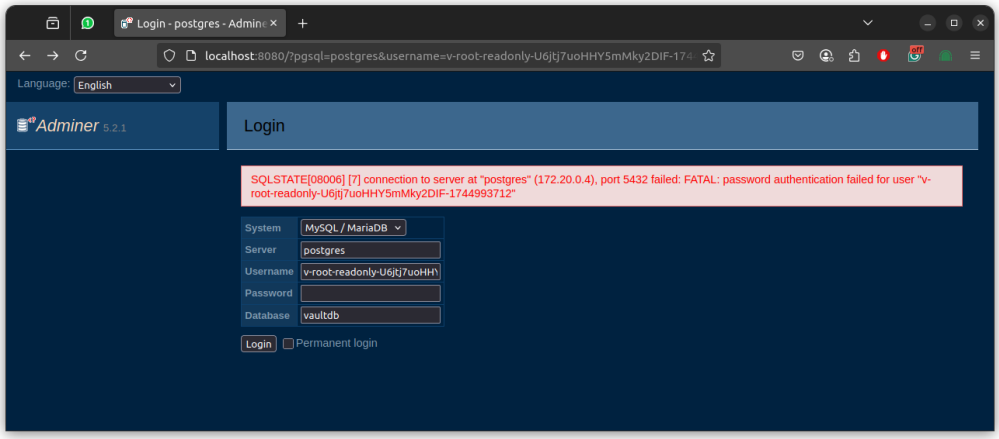
Optional: Revoke the Dynamic Secret

Run below command. Don't forget to replace <<lease_id>> to the real id you have received from the step 6.

```
vault lease revoke <<lease_id>>
```

This will instantly disable the secret.





Conclusion

You just learned how to configure **dynamic secrets** for **PostgreSQL** with Vault.

We:

- Brought up a full stack using Docker.
- Enabled the database engine on a custom path.
- Injected credentials via variables.
- Created and tested a real-time generated database user.
- Explored advanced options like `num_uses` and `{{name}}` templating.

Chapter 17: Dynamic Secrets with MongoDB

MongoDB is a widely-used NoSQL database. In this chapter, we'll configure Vault to dynamically generate credentials for MongoDB, so users or apps never need to know the database root credentials. Vault will create, manage, and revoke them on demand.

Step 0: Clean Up Previous Setup

Before we begin, let's make sure no conflicting containers are running from our previous exercises.

Run:

```
docker compose down
```

This will stop and remove all existing containers. We're starting fresh.

> To speed up the hands-on experience, we've already created the **`docker-compose.yml`** files for you and placed them inside the **`chapter-17`** folder. Make sure to go inside the **chapter-17** folder before executing the **docker compose up -d** command.

Step 1: Docker Compose Setup (Vault + MongoDB)

Let's set up our environment. Create a `docker-compose.yml` file with the following content:

```
services:
  vault-dev:
    image: hashicorp/vault:latest
    container_name: vault-dev
    ports:
      - "8200:8200"
    environment:
      VAULT_DEV_LISTEN_ADDRESS: "0.0.0.0:8200"
    volumes:
      - ../vault-data/config:/vault/config
      - ../vault-data/file:/vault/file
      - ../vault-data/logs:/vault/logs
      - ../:/home/vault/vault-beginner
    cap_add:
      - IPC_LOCK
    command: "server"
  mongo:
    image: mongo:6
    container_name: mongo
    restart: always
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password
```

Start Your Services

Bring up your Vault + MongoDB stack:

```
docker compose up -d
```

Step 2: Attach to the Vault Container

To interact with Vault CLI inside the container, run below command.

```
docker exec -it vault-dev /bin/sh
```

Now you're inside the Vault container.

Step 3: Export the Vault Address

Before using Vault CLI, we need to tell it **where** Vault is running.

```
export VAULT_ADDR=http://localhost:8200
```

This command sets the Vault server address as an environment variable. The CLI uses this to send all your requests to Vault.

Step 4: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

> **Note:** Use the unseal keys you saved during initialization. If those keys are lost, you'll need to reinitialize Vault from scratch.

Step 5: Login to Vault

Login as root

```
vault login <your-root-token>
```

(Replace `<your-root-token>` with your actual root token you have received when you performed the vault operator init.)

Step 6: Enable the Database Secrets Engine on the custom path

Now that you're connected to Vault, enable the database secrets engine on a **custom path** called `mongo-db`:

```
vault secrets enable -path=mongo-db database
```

> We use a custom path to keep secrets engines organized. This way, multiple engines (e.g., MySQL, Postgres, MongoDB) won’t conflict.

Step 7: Configure MongoDB Connection

Set up the database configuration. Replace `<USERNAME>` and `<PASSWORD>` with environment variable references:

```
vault write mongo-db/config/main \
  plugin_name=mongodb-database-plugin \
```

```
  allowed_roles="readonly" \

connection_url="mongodb://{{username}}:{{password}}@m
ongo:27017/admin?authSource=admin" \
  username="root" \
  password="password"
```

Explanation

- `plugin_name`: Uses the official MongoDB plugin.
- `allowed_roles`: Lists roles that are allowed to use this DB connection.
- `connection_url`: Connection string using Vault's placeholders (`{{username}}`, `{{password}}`) that Vault injects dynamically.
- `username/password`: MongoDB root credentials (set via environment).

Step 8: Create a Dynamic Role

Now define the dynamic role `readonly`:

```
vault write mongo-db/roles/readonly \
  db_name=main \
  creation_statements='{ "db": "admin", "roles":
[ { "role": "readAnyDatabase", "db": "admin" } ] }' \
  default_ttl="1h" \
  max_ttl="24h"
```

What's happening?

- `db_name=main`: Tells Vault to use the DB config we defined earlier.

- `creation_statements`: A MongoDB-specific JSON payload that defines what kind of user to create and what roles to assign.
- `default_ttl` and `max_ttl`: Lifetime of the dynamic credentials.

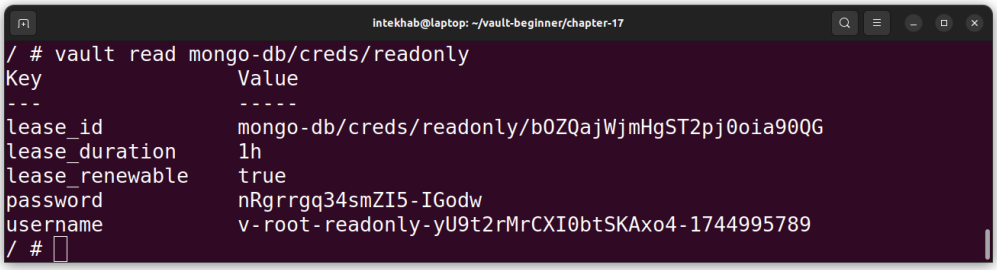
> The dynamic user will get a random username and password with `readAnyDatabase` privileges.

Step 9: Generate Dynamic Credentials

Let's generate a set of dynamic credentials for the `readonly` role:

```
vault read mongo-db/creds/readonly
```

You'll see that the new user has been dynamically created inside MongoDB, and will expire after 1 hour unless renewed.

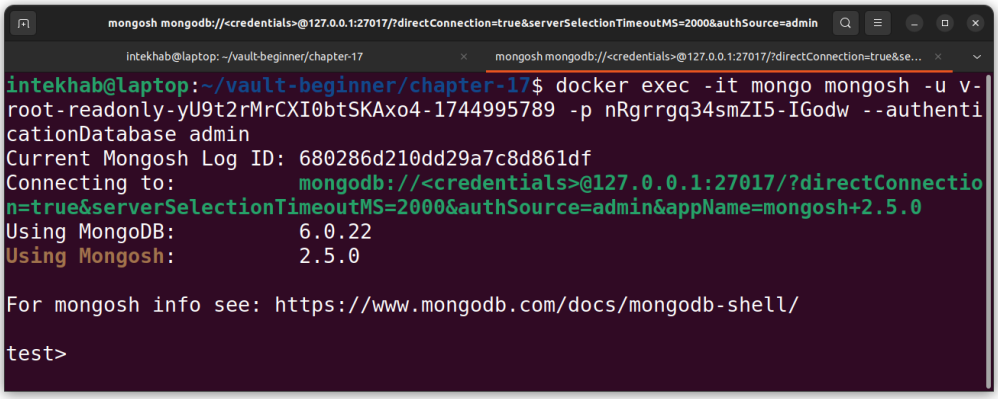


Now let's try to connect on the mongodb server using the newly dynamic secret.

Run below command which will connect us on the mongoDB server running inside our vault stack. Don't forget to replace the <<username>> and <<password>> with the one you have received.

This command connects you to the MongoDB instance running inside our Vault stack using the temporary credentials Vault just generated.

You should see the successful login screen like below.



Want to Verify in Mongo Shell?

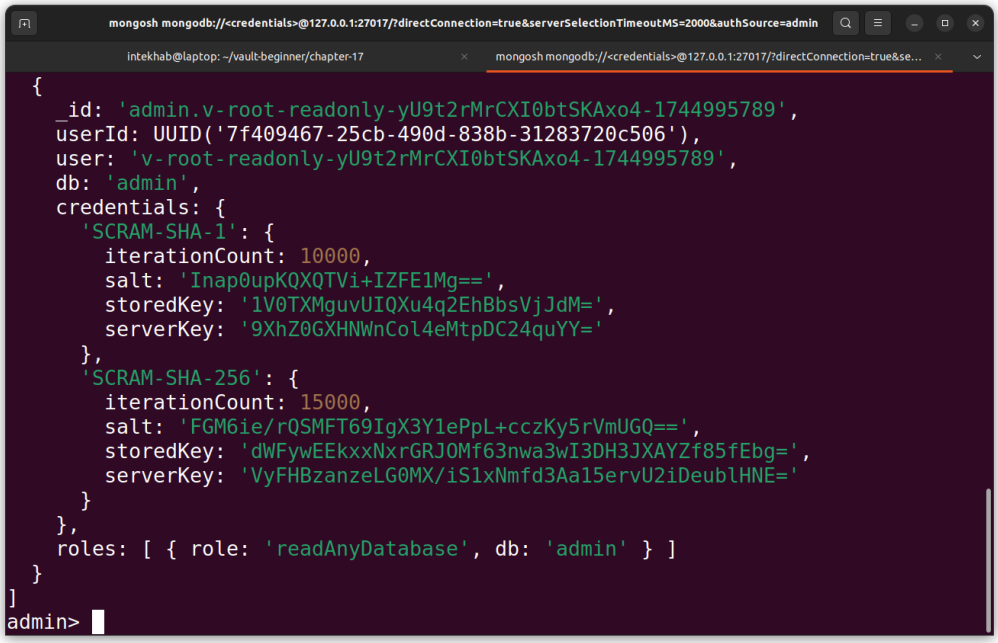
You can connect to the MongoDB as root user:

```
docker exec -it mongo mongosh -u root -p password --
authenticationDatabase admin
```

Then run below command:

```
use admin
db.system.users.find()
```

You'll see a user with a name like `v-root-readonly-....` like below.

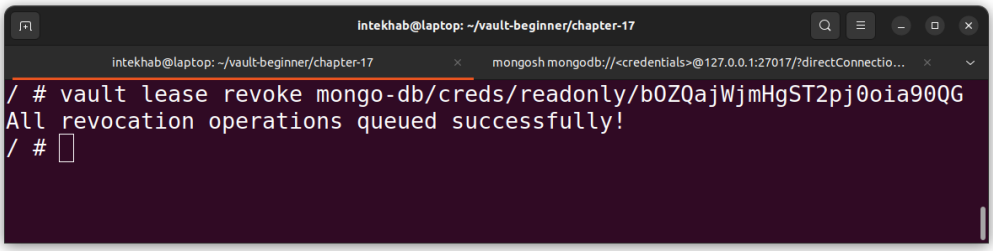


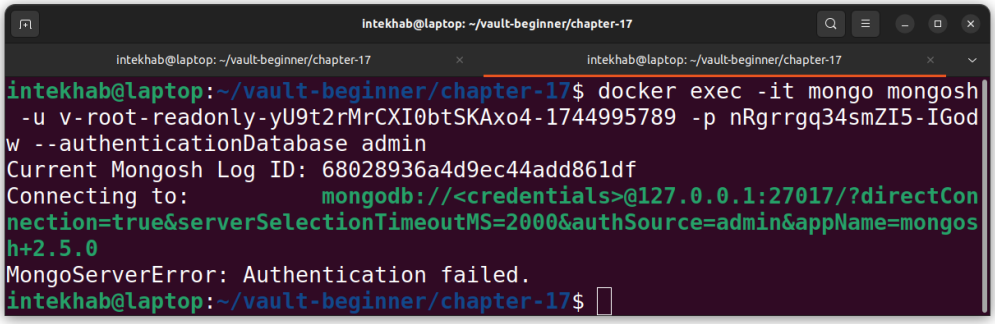
Optional: Revoke the Dynamic Secret

Run below command. Don't forget to replace <<lease_id>> to the real id you have received from the step 9.

```
vault lease revoke <<lease_id>>
```

This will instantly disable the account.





You just learned how to configure **dynamic secrets** for **MongoDB** with Vault.

Chapter 18: Getting Started with the Vault API, Your First Step to Automation

So far, you've been interacting with Vault using the command-line interface (CLI). It's intuitive and ideal for manual tasks. But software doesn't use a terminal—it speaks through APIs. If you want your applications, automation scripts, or microservices to talk to Vault, you need to understand how to use **Vault's HTTP API**.

This chapter marks the transition from **human-driven CLI interaction** to **machine-driven API automation**.

Why Learn the Vault API?

Here’s the reality:

> If you want to automate anything with Vault authenticating, issuing secrets, rotating credentials, you’ll need to use the API.

CLI	HTTP API
Made for people	Made for machines
Simple commands	Standard HTTP requests
Not automation-friendly	Ideal for scripts, apps, pipelines
Manual	Repeatable and scalable

The HTTP API allows Vault to be deeply integrated into your applications and systems. Every single action you’ve taken with the CLI enabling secrets engines, reading secrets, logging in is powered

underneath by an API call. Vault's CLI is simply a wrapper over its HTTP API. Now we're peeling back the wrapper.

In this chapter, we'll explore how to interact with Vault programmatically using HTTP API calls, which is essential for automation and integrating Vault into applications or services. We will specifically work with the **KV V2** secrets engine, which allows you to store and manage secrets as key-value pairs.

Step 0: Clean Up Previous Setup

Before we begin, let's make sure no conflicting containers are running from our previous exercises.

Run:

```
docker compose down
```

This will stop and remove all existing containers. We're starting fresh.

Step 1: Start a Vault Container

Run the following command to start a standalone Vault container, which is sufficient for this exercise.

```
docker run -d --rm \
  -p 8200:8200 \
  -v $(pwd)/vault-data/config:/vault/config \
  -v $(pwd)/vault-data/file:/vault/file \
  -v $(pwd)/vault-data/logs:/vault/logs \
  -v $(pwd):/home/vault/vault-beginner \
  --cap-add=IPC_LOCK \
  --name vault-dev \
    hashicorp/vault:latest server
```

Before starting the Vault container, make sure you're in the root directory of the project, the **vault-beginner** folder.

So far, we've been unsealing Vault using the CLI. In this chapter, we'll switch gears and do everything using the **Vault HTTP API**.

To follow along:

- **Keep your existing Vault container running**.
- **Open a new terminal tab or window** so you can make API requests while Vault is live.
- You're absolutely free to use an API client like **Postman**, **Insomnia**, or **Bruno**—whatever you're comfortable with.
- But to keep things tool-agnostic and beginner-friendly, I'll be using `curl` for all examples in this chapter.

Step 2: Set Environment Variables

Let's set a few environment variables to make our API calls cleaner and avoid repeating values like the Vault address and token every time.

Run the following in your terminal:

```
export VAULT_ADDR=http://127.0.0.1:8200
export VAULT_TOKEN=<<YOUR_ROOT_TOKEN>>
```

Don't forget to replace `<<YOUR_ROOT_TOKEN>>` with the root token you received in **Chapter 3**.

With these set, any `curl` requests we make will be easier to read and maintain.

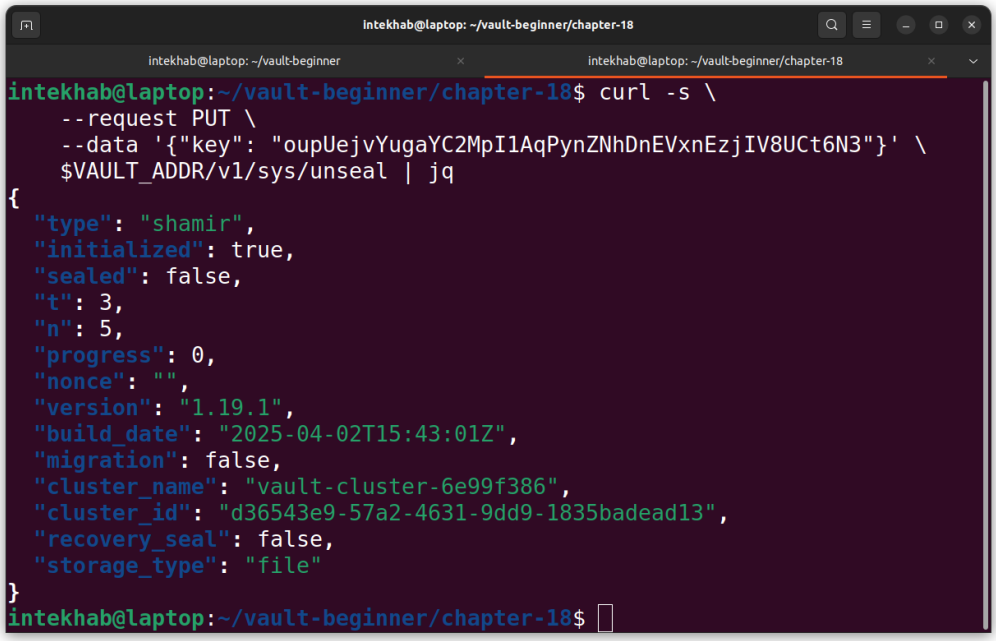
Step 3: Unseal the Vault

Vault requires three unseal keys to unseal. Run the following API request **three times**, replacing the key each time with one of your unseal keys:

```
curl -s \
  --request PUT \
  --data '{"key": "<<UNSEAL_KEY>>"}' \
  $VAULT_ADDR/v1/sys/unseal
```

> In each curl command above, make sure to replace <<UNSEAL_KEY>> with one of the actual unseal keys you received when you initialized Vault in Chapter 3.

After the third successful call, Vault will return `"sealed": false` indicating it's ready to use!



Step 4: Enable the KV V2 Engine via API

Let's enable the **KV V2** secrets engine on the `api-exp` path using the HTTP API. In this step, we will send a POST request to enable the KV V2 engine.

```
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  --request POST \
  --data '{"type": "kv", "options": {"version":
"2"}}' \
  $VAULT_ADDR/v1/sys/mounts/api-exp
```

Explanation:

- `X-Vault-Token: $VAULT_TOKEN`: The authentication token for the Vault API.

- `--request POST`: This sends a POST request to Vault to enable the engine.
- `--data '{"type": "kv", "options": {"version": "2"}}'`: Specifies that we are enabling the KV engine with version 2.
- `$VAULT_ADDR/v1/sys/mounts/api-exp`: The custom path where we want to enable our Kvv2 secret engine.

Once enabled, you'll be able to start storing and retrieving versioned secrets at:

```
api-exp/data/<<ANY_NAME_YOU_LIKE>>
```

Step 5: Create, Read, Delete, and List Secrets via API

Now that the KV V2 engine is enabled, we will go through the basic secret operations such as creating, reading, listing, and deleting secrets.

5.1 Create a Secret

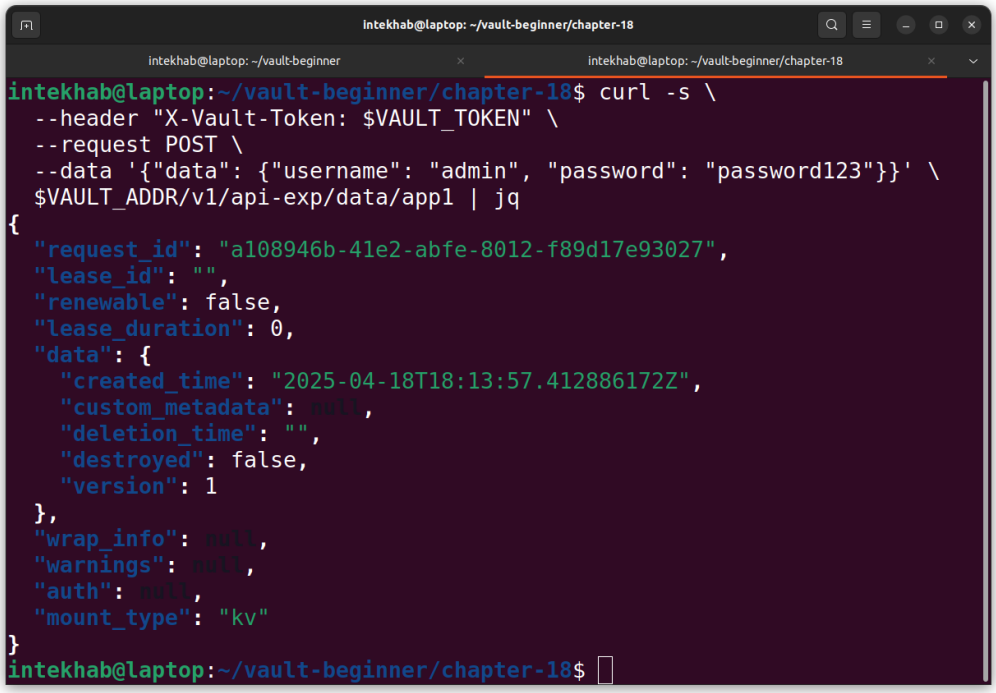
Let's add a new secret under the path `api-exp/data/app1`

```
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  --request POST \
  --data '{"data": {"username": "admin", "password":
"password123"}}' \
  $VAULT_ADDR/v1/api-exp/data/app1
```

Explanation:

- `--request POST`: We are sending a POST request to create a secret.
- `--data '{"data": {"username": "admin", "password": "password123"}}'`: The actual secret data, which is stored as key-value pairs under `data`.
- `$VAULT_ADDR/v1/api-exp/data/app1`: The path where the secret is stored.

On success, you will see the response like below.



5.2 Read a Secret

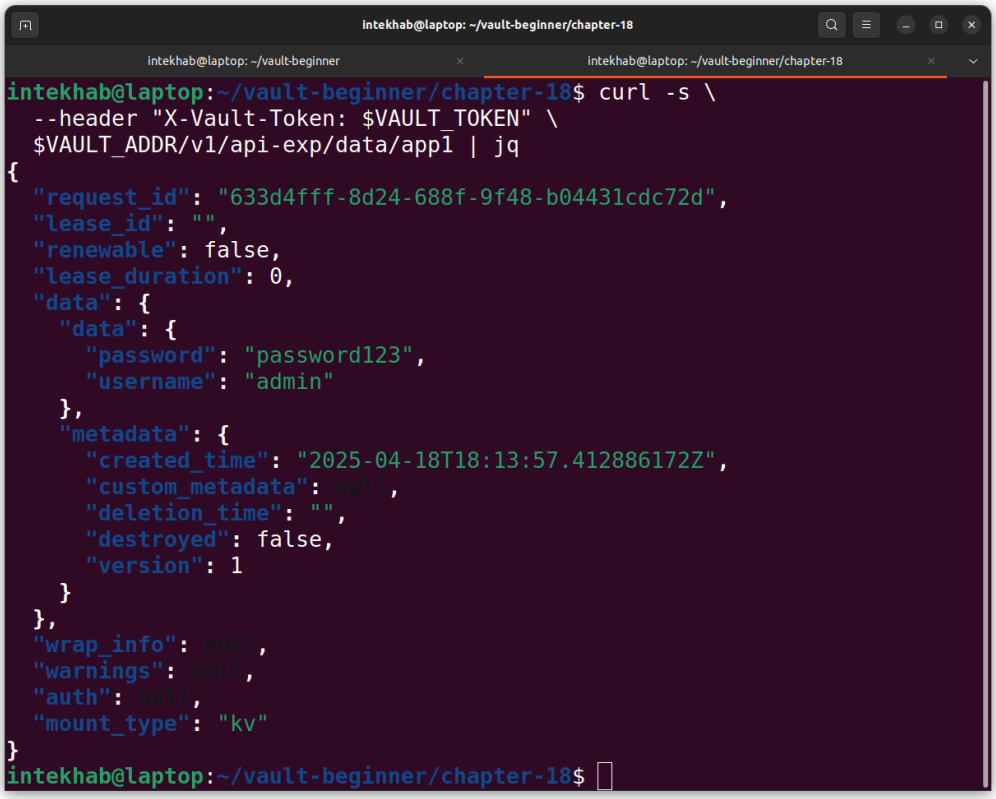
To retrieve the secret stored under api-exp`/data/app1`, use the following GET request:

```
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  $VAULT_ADDR/v1/api-exp/data/app1
```

Explanation:

- `--request GET`: By default, `curl` uses GET for the read operation.

- `$VAULT_ADDR/v1/api-exp/data/app1`: The path where the secret is stored.



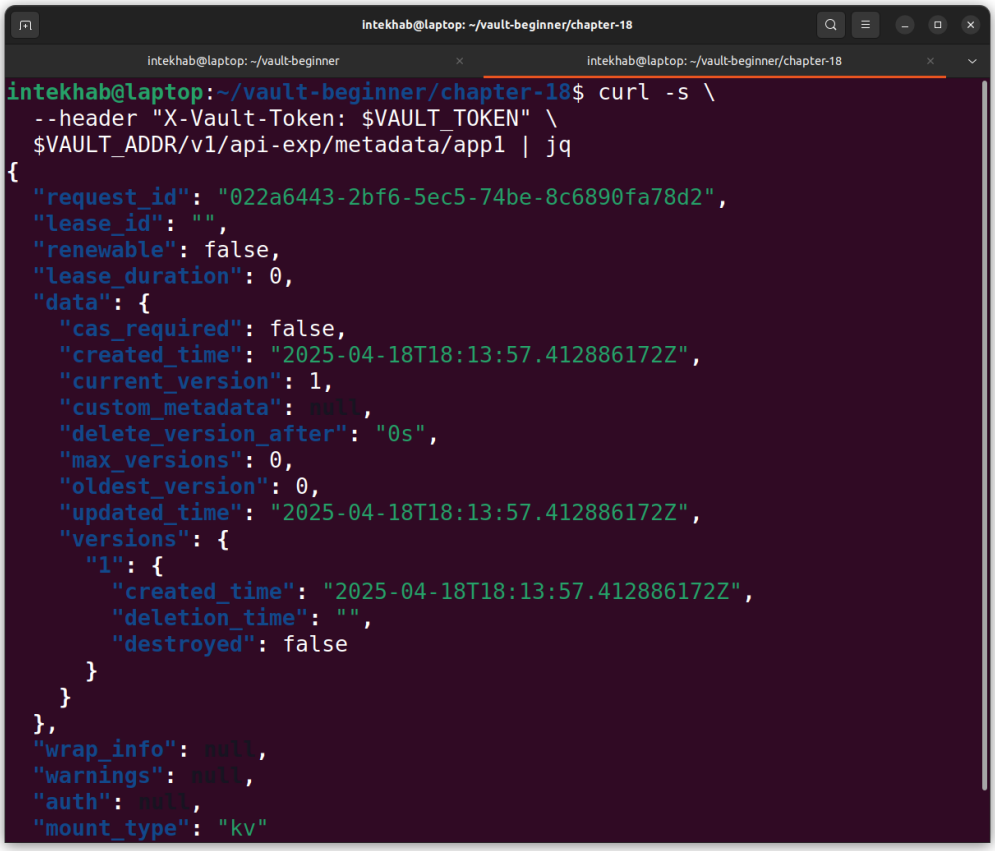
5.3 List Secrets

To list all secrets in the `api-exp` path, use this API call:

```
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  $VAULT_ADDR/v1/api-exp/metadata/app1
```

Explanation:

- `/v1/api-exp/metadata/app1`: This is the path to list all the metadata for secrets stored in the KV V2 engine.



5.4 Delete a Secret

To delete a secret, send a DELETE request:

```
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  --request DELETE \
  $VAULT_ADDR/v1/api-exp/data/app1
```

Explanation:

- `--request DELETE`: The DELETE HTTP method removes the secret from Vault.

- `$VAULT_ADDR/v1/api-exp/data/app1`: The path of the secret to delete.

Step 6: Recover a Deleted Secret

Sometimes you might delete a secret by mistake. Vault provides the ability to **soft-delete** secrets and recover them later.

6.1 Recover a Deleted Secret

If you've deleted a secret, you can recover it by sending a POST request to the `/undelete` endpoint:

```
curl -s \
  --header "X-Vault-Token: $VAULT_TOKEN" \
  --request POST \
  --data '{"versions":[1]}' \
  $VAULT_ADDR/v1/api-exp/undelete/app1
```

Explanation:

- `--data '{"versions": [1]}'`: The version number you wish to recover (you can check which versions are available from the metadata).
- `$VAULT_ADDR/v1/api-exp/undelete/app1/`: The path to the secret that was deleted in this pattern /:secret-mount-path/undelete/:path

6.2 Verify Recovery

Once you've recovered a secret, you can verify its restoration by reading it again:

```
curl -s \
```

```
  --header "X-Vault-Token: $VAULT_TOKEN" \
  $VAULT_ADDR/v1/api-exp/data/app1
```

Summary

In this chapter, we covered the basic operations you can perform with the Vault API on the KV V2 secrets engine. The key points we addressed are:

1. **Enabling the KV V2 Engine**: We used the API to enable the KV V2 secrets engine.
2. **Create, Read, Delete, List Secrets**: We performed the basic operations using the Vault API.
3. **Recovering Deleted Secrets**: We learned how to recover a soft-deleted secret using the API.

Chapter 19: AppRole Authentication Method, Vault Meets Automation

So far, we've operated Vault entirely through human interactions—logging in with tokens, enabling secret engines, and setting policies manually. But in a real-world production environment, this model doesn’t scale. Services, applications, CI/CD pipelines, and background workers need a secure way to authenticate with Vault without human intervention.

That’s where **AppRole** comes in.

AppRole is a secure and flexible authentication method tailored for machines. It provides a way for applications to authenticate to Vault and receive a token with tightly scoped permissions. Unlike static tokens, AppRole uses two distinct pieces of information—**Role ID** and **Secret ID**—to authenticate. This separation of credentials reduces the risk of unauthorized access.

Why AppRole?

Before diving into the implementation, let’s understand the use case for AppRole:

- A CI/CD pipeline needs to pull secrets from Vault to deploy services.
- A containerized microservice needs credentials to connect to a database.

- A backend job needs a time-limited token to access sensitive configuration.

In each of these scenarios, there's no human in the loop. AppRole enables secure, programmatic authentication by issuing short-lived tokens—perfect for automation.

Concepts Behind AppRole

There are a few new terms you'll want to become familiar with:

- **Role ID**: A static identifier assigned to the AppRole. Think of it as the username.
- **Secret ID**: A short-lived, limited-use credential that acts like a one-time password. That means:
    - It has a time-to-live (TTL), after which it becomes invalid.
    - It can be configured to only be used a certain number of times.
    - It helps enforce tighter security, especially in automated environments.
- **AppRole Token**: A Vault token issued after a successful authentication using the Role ID and Secret ID.

You can also fine-tune behaviors such as:

- Token TTL
- Token usage count (via `num_uses`)
- CIDR-bound usage (IP whitelisting)

- Secret ID lifecycle (e.g., automatic rotation or manual generation)

Let's walk through configuring and using the AppRole authentication method in Vault.

In this example, we'll set up a microservice that uses the AppRole authentication method to identify itself and retrieve database credentials. To keep things simple, we'll fetch a static secret from the KV v2 secrets engine instead of using dynamic credentials—which we've already covered. The overall process remains nearly the same for both types of secrets.

Step 1: Prepare Vault Using Docker Compose

Let's start our vault container by running the below command.

```
docker run -d --rm \
  -p 8200:8200 \
  -v $(pwd)/vault-data/config:/vault/config \
  -v $(pwd)/vault-data/file:/vault/file \
  -v $(pwd)/vault-data/logs:/vault/logs \
  -v $(pwd):/home/vault/vault-beginner \
  --cap-add=IPC_LOCK \
  --name vault-dev \
    hashicorp/vault:latest server
```

Step 2: Attach to the Vault Container

```
docker exec -it vault-dev /bin/sh
```

Inside the container, set the Vault address in env variable:

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Step 3: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

Step 4: Login to Vault

Login as root

```
vault login <your-root-token>
```

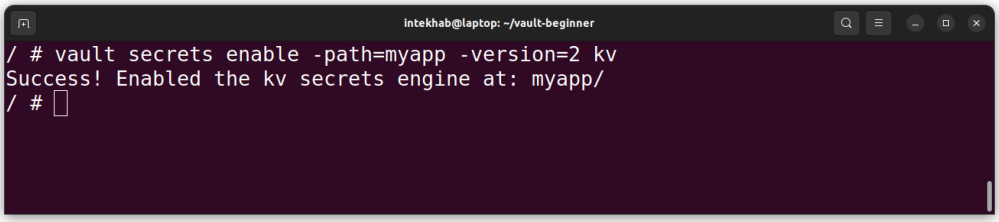
(Replace `<your-root-token>` with your actual root token you have received when you performed the vault operator init.)

Step 5: Enable KV v2 at a Custom Path

Let's first enable the KV v2 secrets engine at a custom path named `myapp`.

```
vault secrets enable -path=myapp -version=2 kv
```

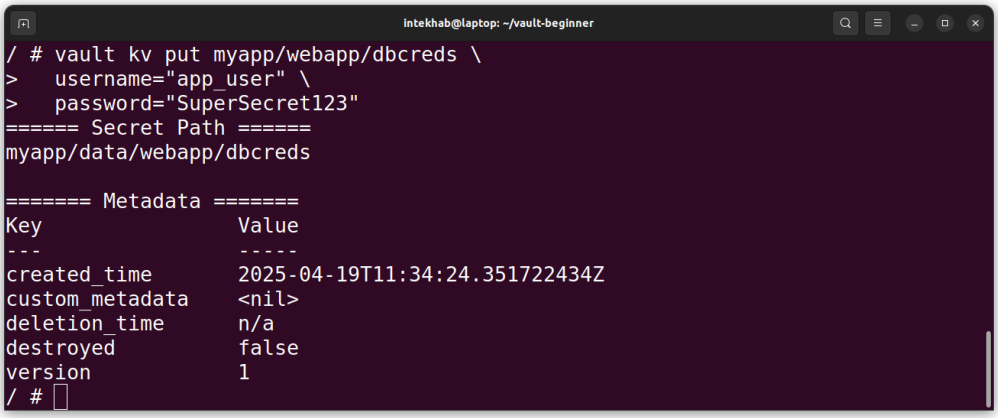
You should see the success message like below.



Step 6: Store a Static Secret in KV v2

Now store a dummy database credential that your microservice will eventually read.

```
vault kv put myapp/webapp/dbcreds \
  username="app_user" \
  password="SuperSecret123"
```



Step 7: Create a Vault Policy

We'll now create a policy that allows read access to only this secret path. Create a policy file `myapp.hcl` with this content:

```
path "myapp/data/webapp/dbcreds" {
  capabilities = ["read"]
}
```

Before applying the policy, make sure you're inside the main project directory, `vault-beginner`**.** This ensures Vault can access the policy file stored in the correct folder structure.

Navigate to the project directory where the myapp.hcl policy file is located:

```
cd /home/vault/vault-beginner
```

This is the location where we've already placed the policy file for you. Apply the policy:

```
vault policy write myapp-reader chapter-19/myapp.hcl
```

Step 8: Enable AppRole Authentication

If AppRole is not already enabled, enable it:

```
vault auth enable approle
```

Step 9: Create the AppRole and Bind the Policy

Create a new AppRole and associate it with the reader policy we have just created. This will give the read only permission on our store secrets on myapp/webapp/dbcreds path.

```
vault write auth/approle/role/myapp \
  token_policies="myapp-reader" \
  secret_id_ttl=60m \
  token_ttl=20m \
  token_max_ttl=60m \
  secret_id_num_uses=5
```

Let's break it down:

- `auth/approle/role/myapp`: This is the path under which the AppRole is registered.
- `token_policies`: The name of the Vault policy that this role's token will inherit.
- `token_ttl`: How long the token will live by default.
- `token_max_ttl`: The maximum lifetime of the token.
- `secret_id_ttl`: The validity period of the Secret ID.
- `secret_id_num_uses`: This Secret ID can only be used 5 times.

Now that the AppRole authentication is set up, any application can authenticate and retrieve the stored credentials. The next step is to generate a **Role ID** and **Secret ID**—these act like the username and password for your application. With them, the application can authenticate to Vault and fetch the secrets it needs.

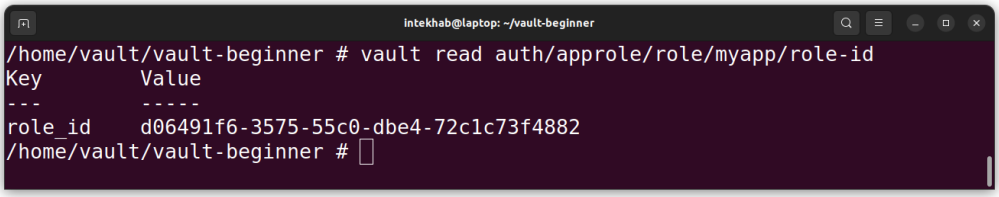
Step 10: Fetch Role ID and Secret ID

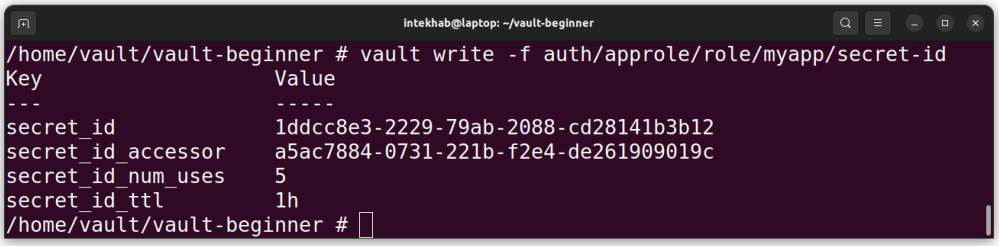
The microservice needs these to log in.

```
# Get Role ID
vault read auth/approle/role/myapp/role-id

# Generate a Secret ID
vault write -f auth/approle/role/myapp/secret-id
```

You'll see an output similar to the one below. Make sure to copy and save both the `role_id` and `secret_id`. we'll need them in the next

steps.





You're now fully set up on the Vault side and ready for a test run. In this step, we'll deploy a pre-built Python-based microservice that connects to Vault using the AppRole authentication method. The service will:

1. Send its Role ID and Secret ID to Vault.
2. Authenticate and receive a Vault token.
3. Use that token to retrieve the secret we stored earlier in the `kv-v2` path.

To keep things simple and streamlined, we'll stop the standalone Vault container and launch both Vault and the microservice using Docker Compose. This automatically creates a shared network between the containers, allowing them to communicate seamlessly—no extra manual setup required.

Let’s get started.

Step 11: Stop the vault standalone container

To stop the vault standalone container, run the below command :

```
docker stop vault-dev
```

Step 12: Configure and start the microservice

To start both containers using Docker Compose and ensure they can talk to each other, follow these steps:

First, navigate to the `chapter-19` folder of your project:

```
cd vault-beginner/chapter-19
```

Inside this folder, you'll find a file named `docker-compose.yaml`. Open it in your preferred editor.

Locate **line 27 and 28**, where you’ll see placeholder values:

```
ROLE_ID: <<WRITE_YOUR_ROLE_ID>>
SECRET_ID: <<WRITE_YOUR_SECRET_ID>>
```

Replace these placeholders with the actual **`role_id`** and **`secret_id`** that you generated earlier.

Once updated, save and close the file. You’re now ready to spin up both containers with proper networking using:

```
docker-compose up
```

This will boot both Vault and the microservice, with environment variables injected and internal networking already handled for you.

Step 13: Attach to the Vault Container

Run the following command to attach to the Vault container:

```
docker exec -it vault-dev /bin/sh
```

Inside the container, set the Vault address in env variable:

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Step 14: Unseal the Vault

Whenever Vault is restarted, it automatically seals itself for security. Before our microservice—or any service—can access secrets, we must unseal Vault first. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

For the curious minds
If you're interested in what's happening behind the scenes at the microservice level, I've included the Python source code in the file named **`microservice.py`** inside the **`chapter-19`** folder. I highly recommend giving it a read to better understand how the

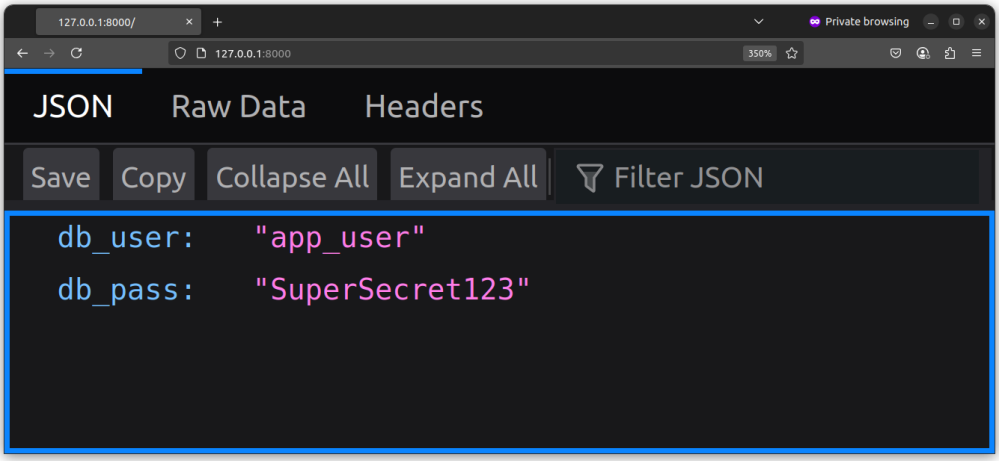
microservice authenticates with Vault, handles the token, and retrieves secrets.

Step 15: Testing the microservice

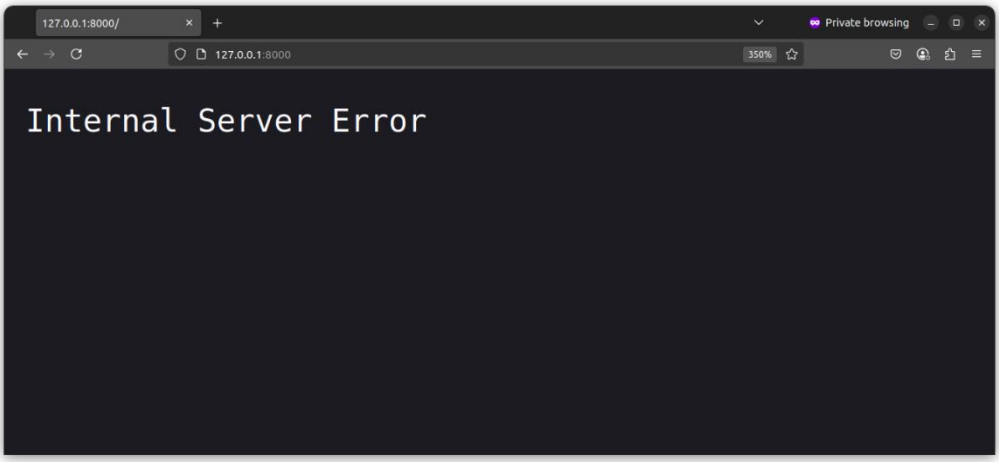
Now that both the Vault and microservice containers are up and running—and Vault has been unsealed—it's time to test if everything is working correctly.

Open your browser or any API client of your choice and navigate to: **http://localhost:8000/**

You should see a JSON response displaying the credentials we stored earlier in **Step 6**.



If you keep refreshing the page, the microservice will continue to return the same credentials. However, after **five successful attempts**, the Role ID and Secret ID will expire. On the **sixth attempt**, authentication will fail, and the microservice will return an internal server error indicating that it could not authenticate with Vault.



Step 16: Shutdown lab

Once you're done with your practice, it's time to shut down our lab setup (Vault and the microservice). To stop and clean up the running containers, simply run:

```
docker compose down
```

This will gracefully stop both services and remove the associated containers and network. Don’t worry, your Vault data remains safe! It’s stored in persistent storage, so it won’t be lost.

Summary

AppRole is your go-to method for secure, programmatic access to Vault. It's flexible, supports automation-friendly token management, and is production-ready. Now your machines have a voice in the Vault ecosystem, without compromising security.

Chapter 20: Vault Agent and Templating, Bridging the Gap for Legacy Apps

Up until now, we've pulled secrets automatically using AppRole, with the application talking directly to Vault. So, you might be wondering—if the credentials are already automated, why do we need Vault Agent at all?

Good question.

Vault Agent becomes essential when you're dealing with **legacy applications** or **third-party tools** that **can’t talk to Vault directly**. These apps often expect credentials to be available in plain files like config files or environment files rather than fetching them dynamically via an API.

Vault Agent acts as a middleman. It runs alongside your application, handles authentication (like AppRole under the hood), pulls secrets from Vault, and writes them to disk in a format your app expects. It can even watch for secret rotations and update those files automatically, all without restarting the application.

In short:
Vault Agent brings Vault’s dynamic secrets to apps that never heard of Vault.

In this chapter we will learn

- What is Vault Agent?
- What is the Vault Template?

- Why and when you should use it
- How to configure Vault Agent
- Running Vault Agent with auto-auth and template rendering
- Live example using KV-v2 engine
- Common pitfalls and real-world use cases

What Is Vault Agent?

Vault Agent is a helper tool provided by HashiCorp Vault that does three major things:

1. **Auto-Authentication**
   Vault Agent can authenticate to Vault using supported auth methods like AppRole, AWS, Kubernetes, etc., and retrieve a client token.
2. **Token Renewal**
   It can keep that token alive using periodic renewal, avoiding the need to re-authenticate repeatedly.
3. **Template Rendering**
   Using Hashicorp Consul Template syntax, Vault Agent can fetch secrets and render them to a file on disk. This is incredibly useful when apps are not Vault-aware but still need secrets.

Vault Agent Configuration Anatomy

Here's what a basic Vault Agent config file looks like:

```
exit_after_auth = false
pid_file = "/tmp/vault-agent.pid"
```

```
auto_auth {
  method "approle" {
    mount_path = "auth/approle"
    config = {
      role_id_file_path = "/etc/vault/role_id.txt"
      secret_id_file_path =
"/etc/vault/secret_id.txt"
          remove_secret_id_file_after_reading = false
    }
  }

  sink "file" {
    config = {
      path = "/etc/vault/token.txt"
    }
  }
}

template {
  source      = "/etc/vault/db-creds.tpl"
  destination = "/etc/secrets/db.env"
}
```

Let's Break It Down:

`exit_after_auth`

- If `true`, Vault Agent will exit after acquiring a token.
- We want a long-running process, so we keep it `false`.

`pid_file`

- Path to a file storing the Agent's process ID.

`auto_auth` Block

This is where we define how Vault Agent should log in.

`method`

- Specifies the auth method—in our case, AppRole.
- The `mount_path` must match where you enabled AppRole in Vault.

`role_id_file_path` & `secret_id_file_path`

- These files must contain the Role ID and Secret ID. These are provided at deployment time (e.g., by CI/CD).

`sink` Block

This tells Vault Agent where to store the generated client token.

```
sink "file" {
  config = {
    path = "/etc/vault/token.txt"
  }
}
```

The app (or Vault Agent itself) can use this token to interact with Vault for other operations.

`template` Block

This is where Vault Agent shines.

You specify:

- `source`: A template file containing the Vault secrets syntax.
- `destination`: Where to write the rendered file.

Example template (`db-creds.tpl`):

```
DB_USER={{ with secret
"myapp/webapp/dbcreds" }}{{ .Data.data.username }}{{
end }}
DB_PASS={{ with secret
"myapp/webapp/dbcreds" }}{{ .Data.data.password }}{{
end }}
```

Once Vault Agent runs, it fetches the secrets and writes them as:

```
DB_USER=app_user
DB_PASS=SuperSecret123
```

Lab Setup: Let’s Make It Real

Step 1: Start Vault Container

We'll reuse the **kv-v2** secrets engine from **myapp** path and **AppRole** setup from the previous chapter. Let’s begin by starting the standalone Vault container using the command below: Make sure you are at the top folder of the cloned repo.

```
docker run -d --rm \
  -p 8200:8200 \
  -v $(pwd)/vault-data/config:/vault/config \
  -v $(pwd)/vault-data/file:/vault/file \
  -v $(pwd)/vault-data/logs:/vault/logs \
  -v $(pwd):/home/vault/vault-beginner \
  --cap-add=IPC_LOCK \
  --name vault-dev \
    hashicorp/vault:latest server
```

Step 2: Attach to the Vault Container

Execute the following command to connect to the running Vault container:

```
docker exec -it vault-dev /bin/sh
```

Once you're inside the container, set the Vault address by exporting it as an environment variable. This ensures the Vault CLI knows where to send requests.

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Step 3: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

Step 4: Login to Vault

Login as root

```
vault login <your-root-token>
```

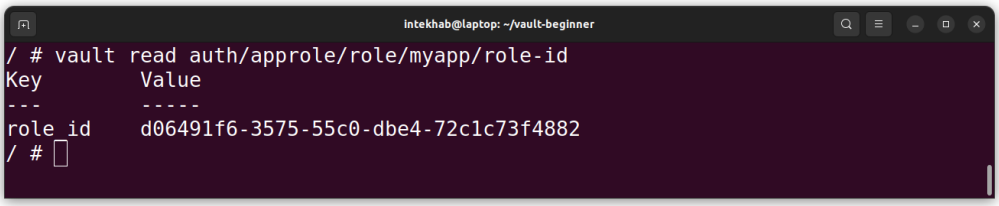
(Replace `<your-root-token>` with your actual root token you have received when you performed the vault operator init.)

Setp 5: Get Role ID

Now let's fetch the **Role ID** by running the following command.
Once you have it, paste just the Role ID into the file located at **chapter-20/role_id.txt**

```
vault read auth/approle/role/myapp/role-id
```

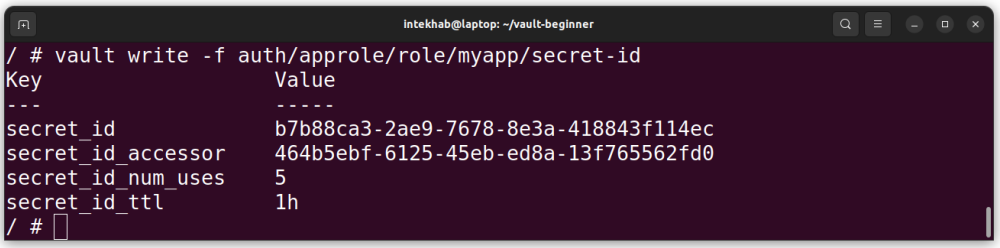
the output will be like below



Step 6: Generate Secret ID

Now let's generate the **Secret ID** by running the following command.
Once you receive it, copy just the **Secret ID** and save it inside **`chapter-20/secret_id.txt`** file.

```
vault write -f auth/approle/role/myapp/secret-id
```



Step 7: Verify db.env and token.txt files

You'll also notice a file named **db.env** and **token.txt** inside the **chapter-20** directory. These files should be empty initially. Once the Vault Agent starts, it will automatically populate these files with the token and credentials autometically from Vault server. Go ahead and open it now to confirm that it's currently empty.

Step 8: Create Credential Template and Agent Config File

First, let's create the credentials template file with below code snippet and store it inside **chapter-20/db-creds.tpl**

```
DB_USER={{ with secret
"myapp/webapp/dbcreds" }}{{ .Data.data.username }}{{
end }}
DB_PASS={{ with secret
"myapp/webapp/dbcreds" }}{{ .Data.data.password }}{{
end }}
```

For the vault agent config file, use below config and save it at **chapter-20/vault-agent.hcl** location.

```
exit_after_auth = false
pid_file = "/tmp/vault-agent.pid"

auto_auth {
  method "approle" {
    mount_path = "auth/approle"
    config = {
      role_id_file_path = "/home/vault/vault-
beginner/chapter-20/role_id.txt"
```

```
      secret_id_file_path = "/home/vault/vault-
beginner/chapter-20/secret_id.txt"
          remove_secret_id_file_after_reading = false
    }
  }

  sink "file" {
    config = {
      path = "/home/vault/vault-beginner/chapter-
20/token.txt"
    }
  }
}

template {
  source      = "/home/vault/vault-beginner/chapter-
20/db-creds.tpl"
  destination = "/home/vault/vault-beginner/chapter-
20/db.env"
}
```

To speed up the hands-on experience, we've already created those files for you and placed them inside the **`chapter-20`** folder.

Step 9: Stop vault standalone container

Now that all the configuration is in place, you'll need the Vault Agent binary to get things going. But don't worry—you don't have to install anything manually. We've already prepped everything for you in a Docker-friendly way.

Instead of running Vault Agent directly, we'll start it inside a container. And to make sure it can talk to the Vault server, we'll

bring both up using Docker Compose—this way, they're on the same network, and communication between them just works.

So first, stop the standalone Vault server container using the below command.

```
docker stop vault-dev
```

Step 9: Start vault stack

Let's start the full stack (Vault Server + Vault Agent) but first go inside the **chapter-20** directory and issue the below command.

```
docker compose up
```

This will start the two containers, one for vault server and other for the vault agent.

When you run this command, Docker starts in **attached mode**—meaning all logs from the services will stream directly to your terminal. Because of that, you won't get your shell prompt back right away.

Let it run as is for couple of seconds.

After a few seconds, you'll likely see an error saying that **Vault is sealed**. That's completely normal.

Now, open another terminal window and attach to the Vault container to **unseal it**.

Step 10: Attach to the Vault Container

```
docker exec -it vault-dev /bin/sh
```

Inside the container, set the Vault address in env variable:

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Step 11: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

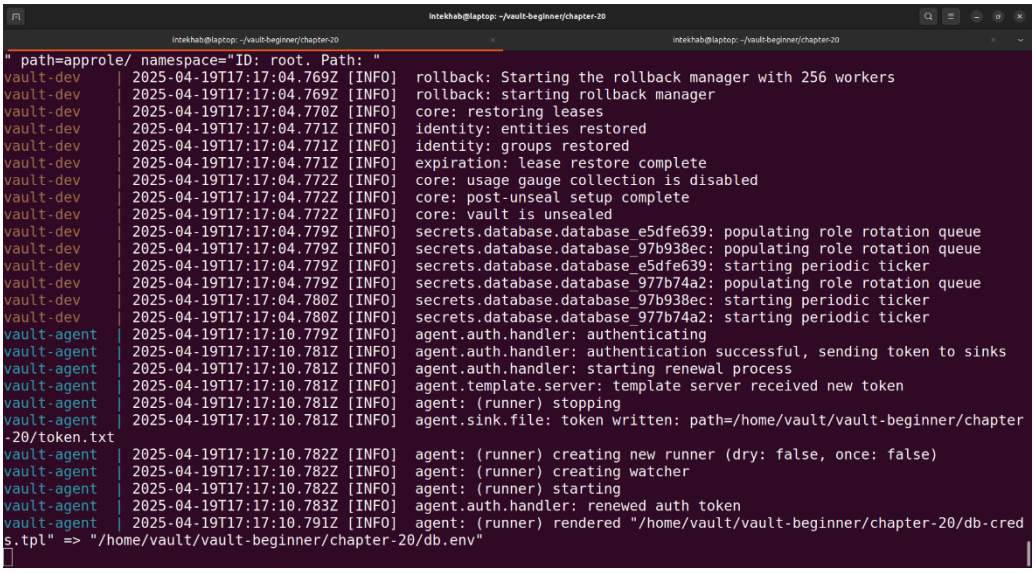
```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

Step 12: Verify the db.env file

Once Vault is unsealed, the **Vault Agent** (running in the first tab) will automatically authenticate using the Role ID and Secret ID, fetch the secrets, and write them to the **`db.env`** file.

You'll also see a log confirmation in the first tab—something like below. Pay attention to the very last line in the screenshot.



That means everything worked! Secrets delivered, no hands required.

Final Result

Your secrets now live inside a db.env like:

```
DB_USER=app_user
DB_PASS=SuperSecret123
```

And your application can source this file without ever knowing Vault exists.

Step 16: Shutdown lab

Once you're done with your practice, it's time to shut down our lab setup (Vault server and vault agent). To stop and clean up the running containers, simply press CTRL + C on the first terminal and after that run below command.

```
docker compose down
```

This will gracefully stop both services and remove the associated containers and network. Don't worry, your Vault data remains safe! It's stored in persistent storage, so it won't be lost.

Real-World Use Cases

- Containerized apps without native Vault integration
- Traditional apps expecting `.env` files
- High-security systems with no outbound internet

Summary:

In this chapter, you leveled up your Vault automation game.

We explored how **Vault Agent** can remove the manual burden of fetching secrets by automatically authenticating with Vault using AppRole, retrieving secrets, and writing them to a local file in a format your application can use.

You configured:

- A **template** to render secrets from Vault into an environment file (`db.env`)
- A **Vault Agent config** that handles authentication and token management
- A **Docker Compose setup** that runs both Vault and Vault Agent in an isolated network

Once everything was in place, you saw how Vault Agent:

- Automatically logs in using Role ID and Secret ID

- Retrieves secrets from Vault
- Populates the `db.env` file without restarting or manual intervention

This chapter shows the real power of automation—perfect for legacy apps that can't speak to Vault directly, and a great way to keep secrets in sync without writing a line of glue code.

Chapter 21: Transit Secrets Engine - Encryption as a Service

Vault isn't just a secure storage locker for your secrets. It can also act as a cryptographic service encrypting and decrypting data without ever storing it. This is the superpower of the **Transit Secrets Engine**. With it, Vault offers **encryption as a service**: your application sends data, Vault encrypts or decrypts it, and your application handles the storage.

Vault never saves your data. It just performs the transformation.

In this chapter, we'll go in-depth on:

- Setting up the Transit engine.
- Creating encryption keys.
- Encrypting and decrypting data.
- Signing and verifying data.
- Real-world use cases.

Why Transit?

Before you reach for client-side encryption libraries, here's what Vault's Transit engine brings to the table:

- **Centralized key management**: Rotate, revoke, and control keys from a single place.
- **No data persistence**: Vault never stores plaintext or ciphertext. It simply encrypts/decrypts on request.

- **Secure APIs**: Your app doesn't need to manage keys, just interact with Vault.
- **Compliance-friendly**: Audit logs track access to keys and operations performed.

Ideal for:

- Tokenization of sensitive fields (e.g card numbers, Customer PII etc etc).
- Database field-level encryption.
- Signing and verifying data blobs.

Let’s do some hands-on practice.

Step 1: Start Vault Container

Let’s begin by starting the standalone Vault container using the command below: Make sure you are at the top folder of the cloned repo.

```
docker run -d \
  -p 8200:8200 \
  -v $(pwd)/vault-data/config:/vault/config \
  -v $(pwd)/vault-data/file:/vault/file \
  -v $(pwd)/vault-data/logs:/vault/logs \
  -v $(pwd):/home/vault/vault-beginner \
  -e VAULT_ADDR="http://localhost:8200/" \
  --cap-add=IPC_LOCK \
  --name vault-dev \
    hashicorp/vault:latest server
```

Step 2: Attach to the Vault Container

Execute the following command to connect to the running Vault container:

```
docker exec -it vault-dev /bin/sh
```

Once you're inside the container, set the Vault address by exporting it as an environment variable. This ensures the Vault CLI knows where to send requests.

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Step 3: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

Step 4: Login to Vault

Login as root

```
vault login <your-root-token>
```

(Replace <your-root-token> with your actual root token you have received when you performed the vault operator init.)

Step 5: Enable the Transit Engine

Let's enable the Transit secrets engine using its default mount path. Run the following command inside the Vault container:

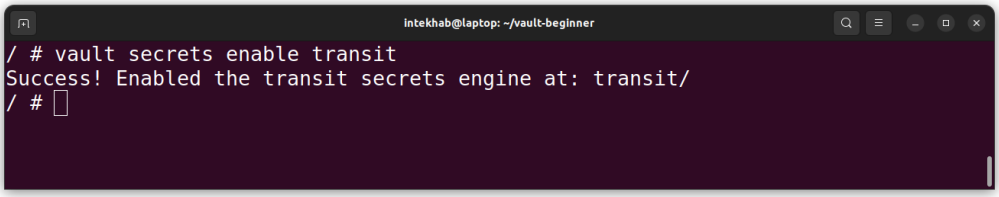
```
vault secrets enable transit
```

This mounts the Transit engine at the default path **transit/**, allowing us to use Vault for cryptographic operations like encryption, decryption, and key rotation—without storing any data.

You can also mount it at a custom path:

```
vault secrets enable -path=enc transit
```

You will see the output like below.



Make sure to use the `transit/` path in all the subsequent commands throughout this chapter. If you've chosen to mount the Transit secrets engine at a custom path instead, don't forget to update the commands accordingly to reflect your specific mount path.

Step 6: Create an Encryption Key

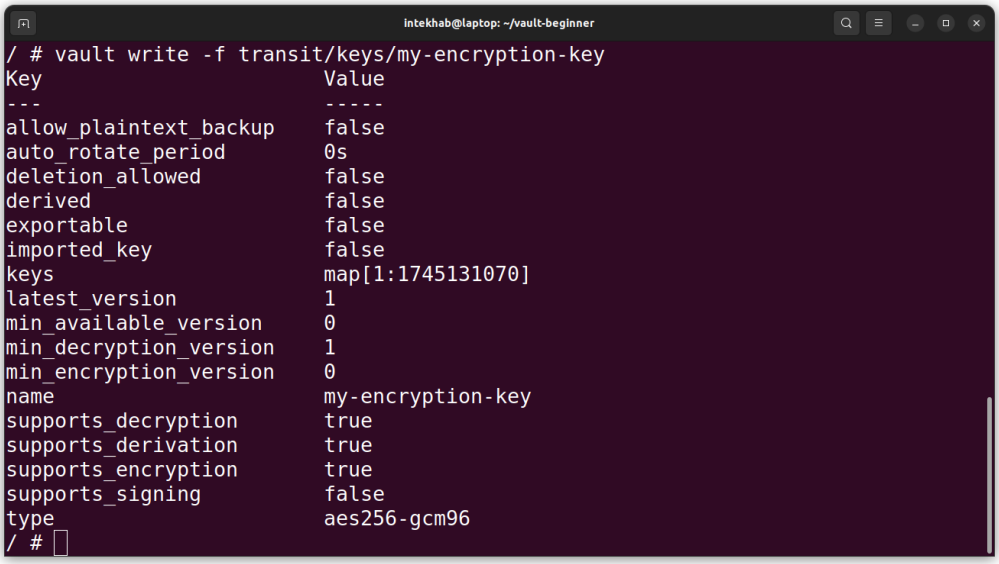
Now that the Transit secrets engine is enabled, it's time to create a named encryption key that Vault will use to perform cryptographic operations. Think of this as defining a logical key inside Vault. Vault manages the actual key material for you behind the scenes. This key will be used to encrypt and decrypt data via the API without exposing the raw key to your application. You can give the key any name (e.g., `customer-data`), and Vault will maintain all the versioning, rotation, and security controls for it. To create it, simply run the below command.

```
vault write -f transit/keys/my-encryption-key
```

Here's what it does:

- Creates a key named `my-encryption-key`.
- The key will be used to encrypt and decrypt data.
- You can create multiple keys, each for a different purpose or app.

Once the command runs successfully, you'll see the configuration details for the newly created key, including its settings and available options, as shown below.



Encrypting Data

Now that your encryption key is ready, let's put it to use. Vault's Transit Secrets Engine doesn't store any data—it's purely for cryptographic operations. That means you send your plaintext data to Vault, and it responds with ciphertext, which you can safely store in your database or elsewhere. This separation of concerns keeps your application logic simple while offloading the complexity and risk of encryption to Vault. To encrypt, you'll pass your plaintext as base64-encoded input along with the key name, and Vault will return an encrypted value you can store with confidence.

```
vault write transit/encrypt/my-encryption-key \
  plaintext=$(echo -n "MySecretMessage" | base64)
```

Explanation:

- Vault expects `plaintext` as base64-encoded string.
- The encrypted result will be returned as `ciphertext`.

Example output:

```
{
  "data": {
    "ciphertext": "vault:v1:IoCzjV..."
  }
}
```

Decrypting Data

Decrypting is just as straightforward as encrypting—Vault handles the heavy lifting for you. When you need to retrieve and use your data, simply send the previously encrypted ciphertext back to Vault, along with the name of the encryption key used. Vault will verify the request, decrypt the data, and return the original plaintext (base64-encoded). Since Vault doesn't store your data, only the encryption keys, this keeps your system lightweight and secure.

```
vault write transit/decrypt/my-encryption-key \
  ciphertext="vault:v1:IoCzjV..."
```

Output:

```
{
  "data": {
    "plaintext": "TXlTZWNyZXRNZXNzYWdl"
  }
}
```

Decode the base64 output:

```
echo "TXlTZWNyZXRNZXNzYWdl" | base64 -d
```

Output:

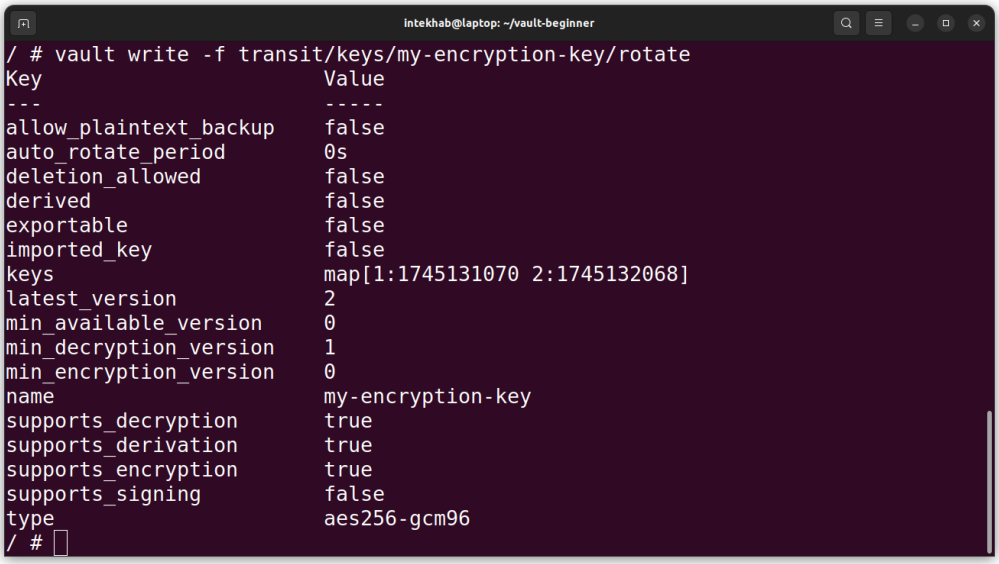
```
MySecretMessage
```

Key Rotation

Key rotation is one of the most powerful features of Vault's Transit engine. Over time, rotating your encryption keys enhances security by limiting the exposure window of any single key. Vault makes this process painless. When you rotate a key, it generates a new version but retains the old ones for decrypting existing data—so nothing breaks. Your app doesn't need to re-encrypt older data unless you explicitly want to rewrap it with the latest key version. This versioned approach gives you the flexibility to keep things secure without disrupting your workflows. And yes, all key versioning and rotation history is fully auditable.

Use below command to rotate your key

```
vault write -f transit/keys/my-encryption-key/rotate
```



In the output, take note of the **`latest_version`** field—it should now show the value **`2`**. This means Vault has generated a second version of the key for that path. From this point on, all new encryption operations will use version 2 by default, while Vault will still retain the previous version(s) to support decryption of older data.

When you rotate a key in Vault, previously encrypted data doesn't become obsolete or unreadable. Vault keeps all older key versions so it can still decrypt any data encrypted using those earlier versions. For example, if your data was encrypted with key version 1, Vault will automatically detect that and use the right key version to decrypt it even after a rotation.

You might have noticed the encrypted string begins with a prefix like **`vault:v1:`**. This is Vault's way of tagging the encryption version used for that data. The **`v1`** part indicates that version 1 of the key was used during encryption. This prefix helps Vault

determine which version of the key to use when decrypting, ensuring everything stays consistent and backwards-compatible.

Absolutely. Here's a refined, instructional book-style version of that explanation:

Step 7: Creating a Transit Key for Digital Signatures

Not all keys are meant for encryption and decryption. In many systems, data integrity and authenticity are just as important. Sometimes even more so. Vault supports this through the use of signing keys, and for that purpose, it provides support for asymmetric key types such as `ed25519`.

The `ed25519` key type does not support encryption or decryption. Instead, it is designed specifically for signing operations. When you use an `ed25519` key with the Transit secrets engine, Vault enables your applications to sign arbitrary data. This allows external systems or services to later verify the signature using a public key, confirming both the origin of the message and that its contents have not been altered.

This is particularly useful in scenarios where you need to:

- Sign structured data such as tokens or audit events.
- Generate digital signatures for messages or payloads.
- Prove the origin of a request in distributed systems.
- Integrate with clients or services that require signature-based verification.

The private signing key never leaves Vault. Signing operations are performed securely within the Vault server, minimizing the risk of key exposure. Vault can also expose the corresponding public key, which can be distributed safely to clients for verification purposes.

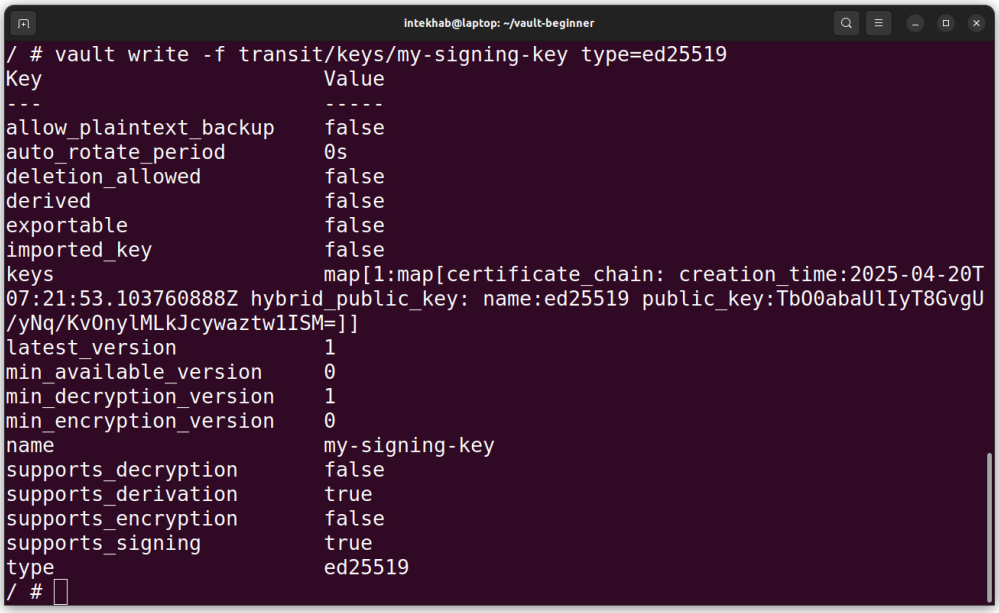
In short, if your goal is to **prove who sent the data and that it wasn't modified**, rather than to encrypt or decrypt it, you should use a key type such as `ed25519`. This ensures cryptographic assurance of authenticity, backed by the secure storage and controlled access that Vault provides.

We'll now proceed to create an **`ed25519`** key and use it to sign and verify data.

Execute the below command to start the creation of the signing key.

```
vault write -f transit/keys/my-signing-key
type=ed25519
```

In this example, we're creating a new key named **`my-signing-key`** with the type set to **`ed25519`**. The output would be like below.



Now that we have a signing key ready, let's walk through how to use it for digital signing and verification.

Signing Data

Signing data using Vault's Transit engine allows you to generate a cryptographic signature for a piece of data without ever exposing the private key. It's ideal when you need to verify the integrity or authenticity of a message, payload, or file.

Before sending your data to Vault, it must be base64-encoded, Vault expects the input in that format to ensure safe transport over the API.

Once encoded, you pass it to the Transit engine, and Vault returns a cryptographic signature. This way, your private keys stay securely tucked away inside Vault, and your application can confidently validate signed data without ever needing direct access to the keys.

Run the below command to sign your data

```
vault write transit/sign/my-signing-key \
  input=$(echo -n "important_data" | base64)
```

You'll get a signature like:

Verifying a Digital Signature

Once data has been signed using a Vault Transit key—such as one created with **`type=ed25519`** you can verify the authenticity of that signature using the **`transit/verify`** endpoint. This step is crucial when you want to ensure that the data has not been tampered with and that it was indeed signed using a trusted key.

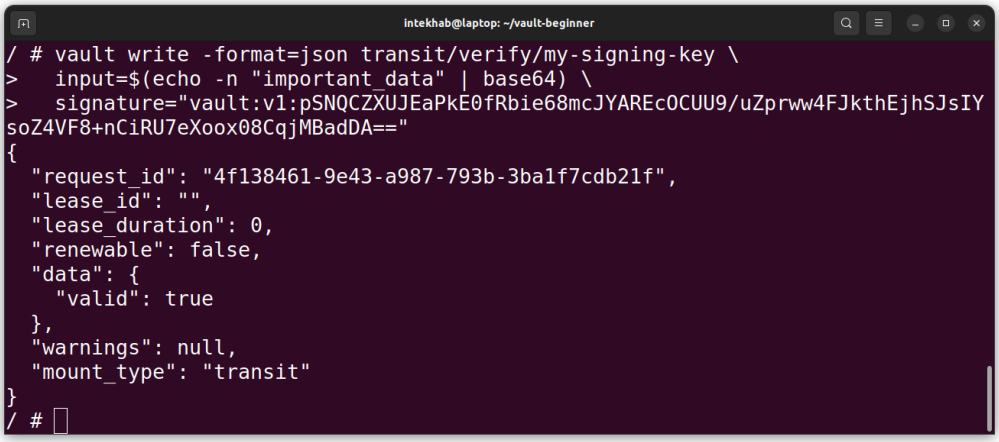
During verification, Vault takes the original input data and the provided signature and checks if they match using the corresponding public key. If the signature is valid, Vault responds with **`valid: true`**, giving you confidence in both the integrity and origin of the data.

Just like signing, the input data must be **base64-encoded** before sending it to the API. This process is often used in multi-service systems to verify tokens, messages, or sensitive payloads where security and trust are critical.

Run the below command to verify the signature

```
vault write -format=json transit/verify/my-signing-
key \
  input=$(echo -n "important_data" | base64) \
  signature="vault:v1:pSNQC...."
```

This returns:



Step 8: Access Control with Policies

Throughout this chapter, we've seen how powerful the Transit Secrets Engine can be. We've created encryption keys, encrypted and decrypted data, rotated those keys, signed payloads, and verified signatures. Each of these operations—while technically simple—must be strictly governed, especially in production environments where multiple services interact with Vault.

This is where **Vault policies** come in.

Vault doesn't just assume a service should be able to use a key simply because it exists. Instead, every capability **`encrypt`**,

`decrypt`, **`sign`**, **`verify`**, **`rotate`**—is exposed through a specific API endpoint. And access to each of these endpoints must be **explicitly allowed** via policy.

Let's connect this to what we've done so far:

- When we used the key to **encrypt** data, Vault hit the endpoint **`transit/encrypt/my-encryption-key`**. To allow this action, a policy must include a rule granting `update` capability on that path.
- For **decryption**, we used **`transit/decrypt/my-encryption-key`**, which again requires its own `update` permission.
- When we rotated the key version using **`transit/keys/my-encryption-key/rotate`**, that operation needed `update` access on a different path.
- Similarly, **signing** with an `ed25519` key involved calling **`transit/sign/my-signing-key`**, and verifying used **`transit/verify/my-signing-key`**.

In essence, Vault policies let you **segment access by operation and by key**. This allows you to:

- Give encryption-only access to applications that store data.
- Restrict decryption access to backend services that process sensitive data.
- Allow signing operations only to systems that produce tokens or messages.
- Let verification be publicly accessible if needed, without exposing the signing key.

This separation is not just best practice—it's vital. If an encrypt-only service is compromised, an attacker can't decrypt data, rotate the key, or sign payloads. It's one of Vault's most powerful security principles: **zero privilege without explicit permission**.

So, as you build policies for your applications, think in terms of roles and responsibilities. Who needs to encrypt? Who should decrypt? Who's allowed to rotate or sign? Define these in separate policies, assign them to specific roles, and let Vault enforce the boundaries.

By combining Transit's features with carefully scoped policies, you're not just managing secrets—you're doing it securely, at scale, and with confidence.

Example Policies for Transit Secret Engine

Below are some of the policies you can consider as your starting point. These are designed to reflect the different operations we've performed throughout the chapter and can be tailored based on your application's trust boundaries.

1. Encrypt-Only Policy

This policy allows services to encrypt data, but not decrypt it—ideal for write-only workflows like logging or form submissions.

```
path "transit/encrypt/my-encryption-key" {
  capabilities = ["update"]
}
```

2. Decrypt-Only Policy

Useful for backend processors that need to read and handle encrypted data but shouldn't create or rotate keys.

```
path "transit/decrypt/my-encryption-key" {
  capabilities = ["update"]
}
```

3. Key Rotation Policy

This policy allows only specific users or automated maintenance tools to rotate keys.

```
path "transit/keys/my-encryption-key/rotate" {
  capabilities = ["update"]
}
```

4. Sign-Only Policy

Grants permission to sign payloads with a specific key. This is useful for token generators or internal services creating signed data.

```
path "transit/sign/my-signing-key" {
  capabilities = ["update"]
}
```

5. Verify-Only Policy

Perfect for public-facing or cross-service verification flows. You can safely distribute this policy to services that need to confirm authenticity but not sign or modify data.

```
path "transit/verify/my-signing-key" {
  capabilities = ["update"]
}
```

6. Full Admin Policy for a Key

If you're building a superuser role (use with caution), this gives full control over a single key, including its deletion.

```
path "transit/keys/my-encryption-key" {
  capabilities = ["create", "read", "update",
"delete", "list"]
}
```

These examples are just the starting point. Depending on how your infrastructure is structured, you can mix, match, and scope policies down even further. Vault's flexibility with path-based access control gives you full power to enforce least privilege—down to a single action on a single key.

Real-World Use Cases

1. **Credit Card Encryption**
   Encrypt card numbers before storing in database.
2. **Tokenization**
   Use encryption + encoding to create tokens from sensitive values.
3. **Secrets-as-a-Service**
   Let internal tools ask Vault to encrypt and decrypt sensitive configs.
4. **Digital Signatures**
   Sign data without sharing private keys across services.

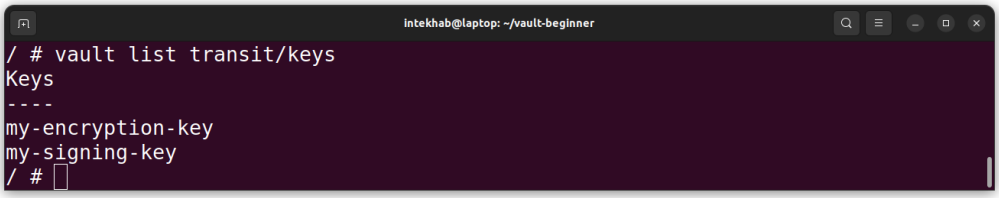
Optional - To List Keys

Once you've started creating encryption keys in the Transit engine, you might want to check which keys are available. Vault makes this easy. You can list all the keys stored in the Transit engine using a simple `vault list` command. This gives you an overview of all active keys managed by Vault—useful for audits, housekeeping, or just keeping track of what's in play.

Run command:

```
vault list transit/keys
```

the output would be like below.



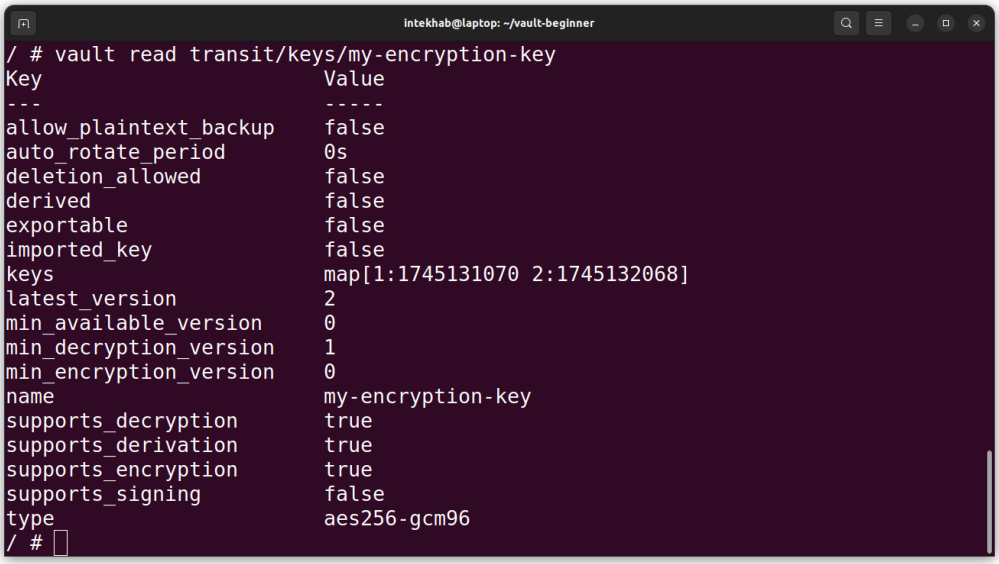
Optional - To Get Key Details

Need to know more about a specific key—like its type, creation time, or the number of key versions? Use the `vault read` command to fetch the key's metadata. This shows you critical information including whether the key supports deletion, whether it's set to auto-rotate, and more. It's especially handy for understanding key lifecycle and verifying that everything's configured as expected.

Run command:

```
vault read transit/keys/my-encryption-key
```

The output would be like below.



Summary

Vault's Transit engine is a powerful way to perform cryptographic operations without exposing or managing raw keys. It decouples your app from key management complexity while offering fine-grained control and auditing.

Chapter 22: Understanding Vault Audit Devices

In a secure system, knowing who did what is as important as protecting what can be done. Vault, being a system of secrets, takes this seriously.

Vault doesn't store an activity history by default. To know what happened inside your Vault, who accessed what secret, when, and how, you must enable **audit devices**.

Think of audit devices as Vault’s black box recorder.

What Are Audit Devices?

An **audit device** in Vault is a configured sink where every request and response to the Vault server gets logged, before and after Vault handles it.

Key Characteristics:

- Immutable and append-only
- Logs both request **and** response payloads
- Logs are **not stored in Vault**, they go to an external system
- Each configured device receives a full audit log entry

Audit logging is **disabled by default**. You must explicitly enable it.

Why Enable Audit Logging?

- **Security visibility**: Know who accessed which secrets
- **Compliance**: Regulatory requirements like SOC2, ISO 27001, HIPAA
- **Forensics**: Investigate a leaked secret or suspicious access
- **Monitoring**: Feed logs into SIEM systems for real-time alerting

Vault does not provide a built-in log viewer or dashboard, you ship the logs elsewhere for storage and analysis.

Types of Audit Devices

Audit Device	Description
File	Writes logs to a local file. Easy for standalone servers or testing.
Syslog	Forwards logs to the system's syslog service (e.g., journald, rsyslog).
Socket	Sends logs to a Unix socket. Often used with log shippers.
HTTP	Sends logs as HTTP POST requests to a remote service. Useful for centralized logging pipelines.
Kafka (Plugin)	Enterprise plugin that streams logs into Kafka.
Datadog (Plugin)	Community or enterprise plugin that sends to Datadog.

For this chapter, we'll use the **file audit device**, as it's the most common starting point.

Audit Log Format

Each log entry is a **JSON object** with fields looks like:

```
{
  "time": "2025-04-14T17:22:00.123456Z",
  "type": "request",
  "auth": {
    "client_token": "hmac-sha256:...",
    "accessor": "hmac-sha256:...",
    "display_name": "approle"
  },
  "request": {
    "id": "b9b3e92b-...",
    "operation": "read",
    "path": "secret/data/production/api-key",
    "remote_address": "10.0.0.15"
  }
}
```

HMAC Protection: Vault never writes secrets or plaintext tokens to logs. Sensitive fields (like token, accessor, paths, and response data) are **hashed with HMAC**. Even if the logs leak, they don't leak secrets.

Let’s set up our lab before we start our hands-on exercises.

Step 1: Start Vault Container

Let’s begin by starting the standalone Vault container using the command below: Make sure you are at the top folder of the cloned repo.

```
docker run -d \
```

```
  -p 8200:8200 \
  -v $(pwd)/vault-data/config:/vault/config \
  -v $(pwd)/vault-data/file:/vault/file \
  -v $(pwd)/vault-data/logs:/vault/logs \
  -v $(pwd):/home/vault/vault-beginner \
  -e VAULT_ADDR="http://localhost:8200/" \
  --cap-add=IPC_LOCK \
  --name vault-dev \
    hashicorp/vault:latest server
```

Step 2: Attach to the Vault Container

Execute the following command to connect to the running Vault container:

```
docker exec -it vault-dev /bin/sh
```

Once you're inside the container, set the Vault address by exporting it as an environment variable. This ensures the Vault CLI knows where to send requests.

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Step 3: Unseal the Vault

Vault is sealed by default for security. To unseal it, you need to provide at least three unseal key shares from the initialization step in **Chapter 3**.

Run the following command **three times**, each time providing **one unseal key** when prompted:

```
vault operator unseal
```

Paste a different unseal key each time. After the third key, Vault will be unsealed and ready to use.

Step 4: Login to Vault

Login as root

```
vault login <your-root-token>
```

(Replace <your-root-token> with your actual root token you have received when you performed the vault operator init.)

Step 5: Pick a secure path to store logs

Before enabling an audit device, Vault needs a location to store the generated logs. Let's begin by selecting a directory for this purpose. In our case, we'll use **`/vault/logs`** as the designated path **inside the Vault container**.

> Outside the container, this directory is mapped to `vault-beginner/vault-data/logs` on your local machine.

Make sure this directory exists and is writable by Vault before proceeding. This setup ensures that audit logs persist even if the container is restarted.

Step 6: Set the Correct Folder Permissions

To allow Vault to write audit logs to the `/vault/logs` directory, we need to ensure that it has the proper ownership inside the container.

Attach to the Vault container, once inside the container, run the following command to change the ownership:

```
chown vault:vault /vault/logs
```

This sets the Vault user and group as the owner of the log directory, ensuring Vault has the necessary write permissions. After this step, you're ready to enable the audit device.

Important: Logs are append-only. Never delete or edit them manually. Rotate using logrotate.

Step 7: Enabling the Audit Device

To enable auditing, we instruct Vault to log each request and response to a specified audit device—in our case, a file on disk. To activate this feature, we'll enable the file-based audit device and point it to the `/vault/logs/audit.log` file inside the container.

Once enabled, Vault will start appending structured log entries for each API operation. These logs are extremely detailed and include timestamps, request paths, client identity, and response status—making them essential for understanding what's happening inside your Vault cluster.

```
vault audit enable file \
    file_path=/vault/logs/audit.log
```

You'll see:

```
Success! Enabled the file audit device at: file/
```

You can verify the enabled audit devices by executing below command:

```
vault audit list
```

Output:

```
Path    Type    Description
----    ----    -----------
file/   file    n/a
```

Step 8: Generate Some Logs

Run a quick command:

```
vault kv get api-exp/app1
```

Then view the audit log:

```
cat /vault/logs/audit.log
```

Each entry will show `type:request` and `type:response` for every command Vault handles.

Please note, using `cat` alone just dumps the file contents, which is okay for small logs but when you're dealing with Vault's audit logs (which can get huge), you'll want more control.

Here are some useful options and alternatives to make your `cat` command more effective when viewing `/vault/logs/audit.log`:

Command / Option	Purpose	Example
`cat /vault/logs/au`	Displays the entire log file (not ideal for large logs).	`cat /vault/logs/au`

Command / Option	Purpose	Example
`dit.log`		`dit.log`
`tail -n <N>`	Shows the last N lines of the log. Good for recent activity.	`tail -n 50 /vault/logs/audit.log`
`tail -f`	Continuously displays new log entries in real-time (live view).	`tail -f /vault/logs/audit.log`
`tail -n <N> -f`	Shows last N lines and follows new logs.	`tail -n 50 -f /vault/logs/audit.log`
`head -n <N>`	Displays the first N lines of the log. Useful for seeing earliest entries.	`head -n 20 /vault/logs/audit.log`
`grep "<keyword>"`	Filters log lines by keyword (case-sensitive).	`grep "transit" /vault/logs/audit.log`
`grep -i "<keyword>"`	Case-insensitive search for keyword.	`grep -i "LOGIN" /vault/logs/audit.log`
`grep -A <N> "<keyword>"`	Shows N lines after a match. Helpful for context.	`grep -A 3 "login" /vault/logs/audit.log`
`grep -B <N> "<keyword>"`	Shows N lines before a match.	`grep -B 3 "login" /vault/logs/audit.log`

Optional: Configure Audit Log Rotation

Vault does not automatically rotate its audit log, which means the log file can grow indefinitely over time. To manage this, you can use the `logrotate` utility to rotate logs safely and efficiently.

Start by creating a logrotate configuration file:

```
sudo nano /etc/logrotate.d/vault-audit
```

Add the following configuration:

```
/vault/logs/audit.log {
  daily
  rotate 7
  compress
  missingok
  notifempty
  create 0640 vault vault
  postrotate
    systemctl kill -HUP vault
  endscript
}
```

Here's a breakdown of what this does:

- **daily**: Rotates the log every day.
- **rotate 7**: Keeps the last 7 log files before deleting old ones.
- **compress**: Compresses older logs to save space.
- **missingok**: Doesn't throw an error if the log file is missing.
- **notifempty**: Skips rotation if the file is empty.

- **create 0640 vault vault**: Creates a new log file with proper permissions.
- **postrotate / endscript**: Sends a `HUP` signal to Vault, prompting it to close the current log and open a new one.

This setup ensures that Vault's audit logging remains clean, controlled, and doesn't fill up your disk over time.

Audit Log Permissions & Security

- Store logs in a tamper-evident location (e.g., remote S3 bucket, read-only volume)
- Use a log forwarder (e.g., FluentBit, Filebeat) to ship logs off-box
- Lock down file access (`chmod 640`, only readable by Vault and log collector)
- Regularly monitor audit logs for unusual paths or access patterns

Recap

Audit devices are Vault’s window into user behavior. They record every request and response interaction without ever logging secrets. You learned:

- What audit devices are and why they matter
- Different types: file, syslog, HTTP, Kafka
- How to configure a file-based audit device
- How to monitor and rotate your logs securely

You should now have full visibility into what happens inside your Vault.

Chapter 23: Revoking and Regenerating the Root Token

In a production Vault deployment, the **root token** is the most powerful credential in the system. It is designed to **bypass all policies**, granting full administrative access. While this makes it essential during setup and bootstrapping, **retaining the root token after initial configuration introduces significant risk**.

In this chapter, we'll cover:

1. Why revoking the root token is strongly recommended
2. How to **revoke the root token** securely
3. How to **regenerate the root token** using quorum
4. Enabling a stable operational authentication method (e.g., `userpass`) beforehand
5. Understanding `mlock`, the Vault data directory, and best practices
6. A complete walkthrough with all commands included

Why Revoke the Root Token?

The root token:

- Has **unrestricted access** to Vault.
- Bypasses policies and controls.
- If leaked or misused, **compromises the entire Vault cluster**.

Root Token in Production = Liability

Best practice:

- Use it to bootstrap Vault.
- Create real users and assign policies.
- Then revoke it.

Regeneration of the root token should be a deliberate, audited, multi-party operation, not a casual fallback.

Pre-Requisite: Enable the `userpass` Auth Method

Before revoking the root token, ensure you have **an alternate administrative login**.

Here, we'll use the built-in `userpass` auth method.

Step 1: Enable `userpass` Auth

```
vault auth enable userpass
```

Step 2: Create an Admin Policy

Create a file `admin-policy.hcl with below content`:

```
path "*" {
  capabilities = ["create", "read", "update",
"delete", "list", "sudo"]
}
```

Create the policy using the admin-policy.hcl:

```
vault policy write admin admin-policy.hcl
```

Ensure you are inside the folder where you saved the admin-policy.hcl

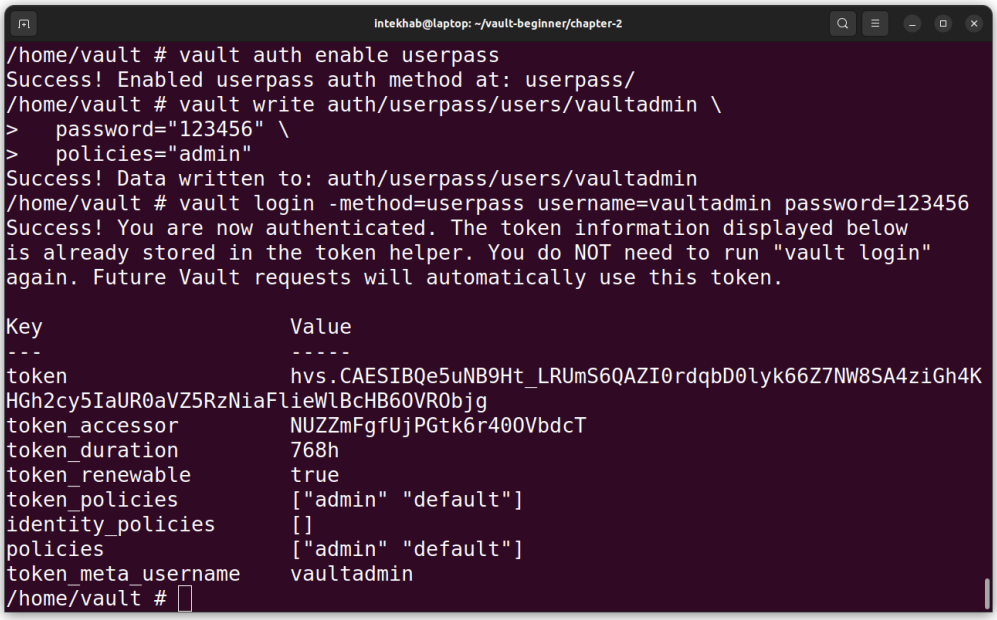
Step 3: Create a Vault Admin User

```
vault write auth/userpass/users/vaultadmin \
  password="123456" \
  policies="admin"
```

Login and verify:

```
vault login -method=userpass username=vaultadmin
password=123456
```

The output of last there commands will be like below.



Now you're ready to retire the root token.

Revoking the Root Token

If you're logged in as root and want to revoke your own token:

```
vault token revoke -self
```

Or to revoke a specific root token:

```
vault token revoke <root_token>
```

> Before revoking, always confirm another fully privileged user or group has operational access to Vault.

Regenerating the Root Token (When Needed)

Vault supports a **secure, quorum-based process** to regenerate the root token. This process ensures that **no single person can generate the root token alone**.

Step 1: Begin Root Token Regeneration Process

Start the root token generation process:

```
vault operator generate-root -init
```

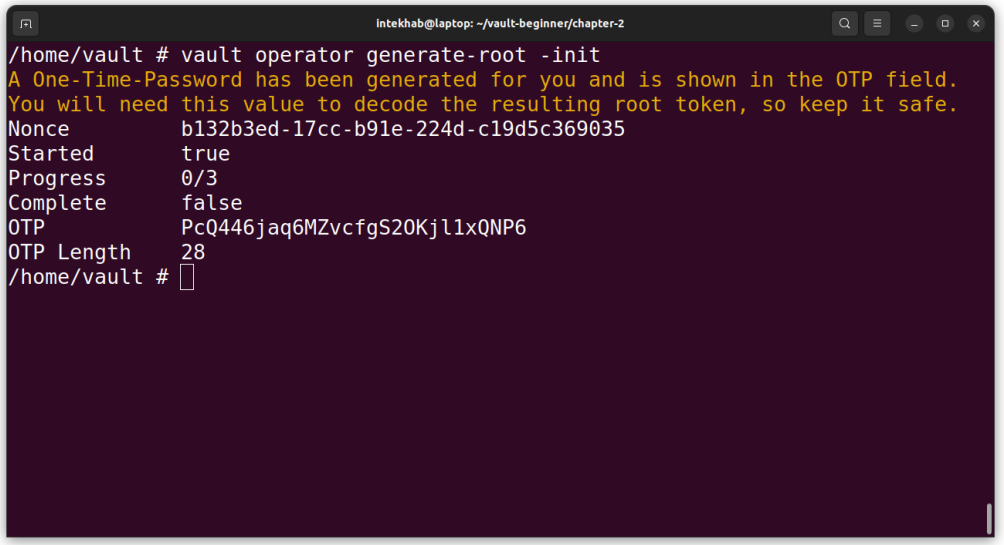
You will receive two important values:

- **One-Time OTP (OTP)**: A short-lived password used to encrypt the new root token.

- **Nonce**: A unique identifier for this regeneration attempt. It must be passed with every unseal key submission.

> **What is OTP and Nonce?**
> The **OTP** is used to **encrypt the root token**. You'll need it at the end to decrypt the result. The **Nonce** uniquely identifies this regeneration attempt. Without it, Vault won't accept unseal key submissions.

Example output:



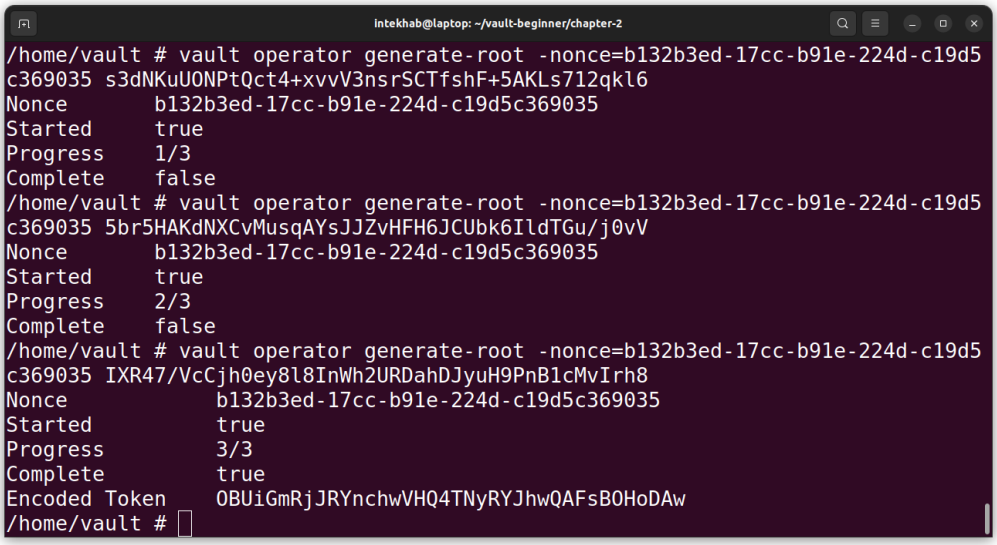
Step 2: Provide Unseal Key Shares (With Nonce)

Each key holder submits their unseal key, **along with the nonce**:

```
vault operator generate-root -nonce=<nonce>
<unseal_key>
```

Repeat this step with each unique key until the configured key threshold is met (e.g., 3 of 5 unseal keys).

Once quorum is reached, Vault outputs the **Encoded Root Token**.



Step 3: Decrypt the Root Token (Using OTP)

Vault returns the token like this:

```
Encoded Token: xxxxxxxxxx
```

Now decrypt the encoded token using the OTP you received earlier:

```
vault operator generate-root -decode <encoded_token>
-otp <One-time-otp>
```

You'll then receive the **actual root token**.

Recap

- The **root token should not be kept around** in production systems.
- Always create admin users and test their login **before revoking** the root token.
- The **root token can be regenerated** using quorum-based approval from unseal key holders.
- Vault's **data directory** and `mlock` settings are critical for performance and security.

Chapter 24: Production deployment of Hashicorp Vault on Ubuntu LTS

In the playground, we spun up Vault with a single command. In production? Not so fast.

Deploying Vault in the real world means locking it down, bootstrapping it correctly, and setting it up to survive power outages, restarts, and mistakes.

This chapter walks you through setting up **HashiCorp Vault on an Ubuntu LTS server**, with or without a domain name, securely and correctly.

What You'll Learn

- Installing Vault from the official repository
- Difference between IP-based and domain-based access
- Vault configuration anatomy and key files
- Importance of memory locking (`mlock`)
- File-based storage with Raft backend
- TLS configuration
- Managing Vault as a systemd service
- Vault initialization, unsealing, and best practices

IP vs Domain, What Changes in Real-World Setup?

Vault is an HTTP API server at its core. It doesn't require a domain name to run, but how you deploy it affects TLS, automation, and best practices.

Criteria	IP-Based Setup	Domain-Based Setup
TLS Certificate	Self-signed or internal CA	Public CA (Let's Encrypt, etc.)
`api_addr` / `cluster_addr`	IP-based (e.g., https://10.0.0.10:8200)	FQDN-based (e.g., https://vault.company.com)
Automation (CI/CD, Vault Agent)	Slightly harder to manage	Easier via DNS resolution
Vault UI Availability	Works but browser warns about TLS	Secure and smooth

You can run Vault with only an IP and a self-signed certificate. But for enterprise setups or developer convenience, it's strongly recommended to use a fully qualified domain name (FQDN) and proper TLS.

Step 1: Install Vault (Official Binary)

Install Vault from HashiCorp's APT repository:

```
sudo apt update && sudo apt install -y gnupg curl
software-properties-common

curl -fsSL https://apt.releases.hashicorp.com/gpg | \
```

```
  sudo gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/hashicorp-
archive-keyring.gpg] \
  https://apt.releases.hashicorp.com $(lsb_release -
cs) main" | \
  sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update;
sudo apt install vault;
```

Step 2: Prepare Folders & TLS

Vault Folders

Vault doesn't create its own directory structure. You must define it:

```
sudo mkdir -p /etc/vault.d /opt/vault/data
/opt/vault/logs;
sudo chown -R vault:vault /opt/vault;
```

- `/etc/vault.d` → All config files go here
- `/opt/vault/data` → Where Vault stores secrets when using integrated Raft storage
- `/opt/vault/logs` → Optional log directory if using file-based audit logs

Vault Data Folder Explained
When using the Raft storage backend, Vault persists all data—including secrets, policies, and leases—inside the `data` directory. It is a critical directory and **must be backed up** regularly using snapshot APIs.

TLS Setup

With IP (Self-Signed)

Use OpenSSL:

```
openssl req -x509 -nodes -days 365 \
  -newkey rsa:2048 \
  -keyout /etc/vault.d/vault.key \
  -out /etc/vault.d/vault.crt \
  -subj "/CN=<<WRITE_YOUR_IP_ADDRESS_HERE>>";

sudo chown vault:vault /etc/vault.d/vault.*;
```

With Domain (Recommended)

```
sudo apt install certbot;
sudo certbot certonly --standalone -d
<<WRITE_YOUR_FQDN_HERE>>;
```

Then:

```
sudo cp
/etc/letsencrypt/live/<<WRITE_YOUR_FQDN_HERE>>/fullch
ain.pem /etc/vault.d/vault.crt;

sudo cp
/etc/letsencrypt/live/<<WRITE_YOUR_FQDN_HERE>>/privke
y.pem /etc/vault.d/vault.key;

sudo chown vault:vault /etc/vault.d/vault.*;
```

Step 3: Vault Configuration Explained

Create a file at `/etc/vault.d/vault.hcl`:

```
ui = true
log_level = "info"

storage "raft" {
  path    = "/opt/vault/data"
  node_id = "vault-node-1"
}

listener "tcp" {
  address       = "0.0.0.0:8200"
  tls_cert_file = "/etc/vault.d/vault.crt"
  tls_key_file  = "/etc/vault.d/vault.key"
}

api_addr = "https://<<YOUR_IP_OR_FQDN_HERE>>:8200"
cluster_addr =
"https://<<YOUR_IP_OR_FQDN_HERE>>:8201"

disable_mlock = false
```

Breakdown of Key Config Items:

Key	Description
`ui = true`	Enables Vault's browser interface
`storage "raft"`	Uses Raft for HA-ready local storage
`node_id`	Unique name of this Vault node
`listener`	Defines network interface and TLS settings
`api_addr`	How clients communicate with Vault (must match cert)
`cluster_addr`	Used for internal cluster gossip
`disable_mlock`	**Set to false in production**; read below

About `mlock` (Memory Lock)

Vault uses in-memory encryption and holds sensitive data in RAM. Linux swaps out memory pages to disk under pressure, which is a **security nightmare** for secrets.

The `mlock` syscall prevents memory from being swapped.

What Happens If You Don't Use `mlock`?

- Vault will log a warning
- Secrets in memory could end up in disk swap
- Regulatory compliance may be violated

Step 4: Create Vault systemd Unit

Create `/etc/systemd/system/vault.service`:

```
[Unit]
Description=HashiCorp Vault
Requires=network-online.target
After=network-online.target

[Service]
User=vault
Group=vault
ProtectSystem=full
ProtectHome=read-only
PrivateTmp=yes
Capabilities=CAP_IPC_LOCK
CapabilityBoundingSet=CAP_IPC_LOCK
ExecStart=/usr/bin/vault server -
config=/etc/vault.d/vault.hcl
```

```
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
LimitMEMLOCK=infinity
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

Then reload and start Vault:

```
sudo systemctl daemon-reexec;
sudo systemctl enable vault;
sudo systemctl start vault;
```

Step 5: Initialize and Unseal Vault

First, set the address (replace with your IP or domain):

```
export
VAULT_ADDR=https://<<YOUR_IP_OR_FQDN_HERE>>:8200;

export VAULT_CACERT=/etc/vault.d/vault.crt
```

Initialize Vault:

```
vault operator init -key-shares=5 -key-threshold=3 >
~/vault.init
```

Unseal Vault (you need 3 keys out of 5):

```
vault operator unseal <key1>
vault operator unseal <key2>
vault operator unseal <key3>
```

Step 6: Security and Hardening Checklist

Measure	Description
TLS Everywhere	Never run Vault without TLS, even locally
Memory Lock	Prevent secrets from hitting disk swap
Firewall	Block everything except port 8200 (and 443 if reverse proxy)
Audit Logging	Enable file-based audit logging
Secrets Backups	Use Raft snapshot API regularly
Limited Root	Avoid running Vault CLI with root privileges
Cert Renewal	Automate TLS cert renewals via cron or timer

Final Validation

If everything went well, visiting:

```
https://<<YOUR_IP_OR_FQDN_HERE>>:8200
```

Should show you the Vault UI.

Recap

In this chapter, you learned how to:

- Deploy Vault on an Ubuntu LTS server with Raft storage
- Secure it using TLS and memory locking
- Understand the Vault folder structure and key config parameters
- Run it as a production-grade service using systemd

- Work with both IP-based and domain-based setups

You now have a Vault server that’s reliable, secure, and production-ready.

Chapter 25: Production Deployment of Hashicorp Vault Using Docker

Docker is everywhere and it's powerful, portable, and consistent. If you're managing microservices or orchestrating infrastructure across environments, you've probably considered using Docker for your Vault deployment.

And yes, it's absolutely viable for **production**.

In this chapter, we'll walk through everything you need to securely run **HashiCorp Vault in production using Docker**:

- Why Docker for Vault
- Production-grade setup
- Static vs dynamic configuration
- Secure persistent storage
- TLS termination
- Deployment with or without a domain name
- Environment variable injection
- Memory locking and tuning

Why Docker for Vault?

Using Docker in production has some advantages:

- Simple packaging and portability

- Works on any host with Docker Engine
- Clean separation of concerns (Vault container, storage, network)
- Works with orchestration tools like Docker Compose, Kubernetes, or Nomad

But **Docker does not make Vault magically production-ready.**

You still need to:

- Set up secure storage
- Handle TLS correctly
- Configure auto-unsealing (if desired)
- Mount proper data volumes
- Harden access and audit

Let's go step by step.

Step 1: Preparing the Host Machine

Install Docker on a **Linux server**, preferably **Ubuntu LTS 24.04 or later**:

```
sudo apt update && sudo apt install -y docker.io
sudo systemctl enable docker
```

Create a working directory:

```
mkdir -p /opt/vault-prod/config /opt/vault-prod/data
/opt/vault-prod/logs
```

We'll mount these into the container later:

- `/config`: Vault configuration file(s)
- `/data`: Vault's persistent storage
- `/logs`: Optional logging output

Step 2: Create a Production Vault Configuration File

Create `/opt/vault-prod/config/vault.hcl`:

```
ui = true
# Storage backend: Raft
storage "raft" {
  path    = "/vault/data"
  node_id = "vault-node-1"
}

# Listener with TLS enabled
listener "tcp" {
  address     = "0.0.0.0:8200"
  tls_cert_file = "/vault/config/certs/vault.crt"
  tls_key_file  = "/vault/config/certs/vault.key"
}

api_addr = "https://<<YOUR_IP_OR_FQDN_HERE>>:8200"

cluster_addr =
"https://<<YOUR_IP_OR_FQDN_HERE>>:8201"

disable_mlock = false

# Log level
log_level = "info"
```

`disable_mlock = false` ensures Vault memory is not swapped to disk. You'll need to allow `CAP_IPC_LOCK` when starting the container.

Step 3: Vault Data Directory

Vault stores everything—secrets, leases, metadata—in the Raft storage directory (`/vault/data`).

Best practices:

- Mount a **dedicated volume** for durability
- Regularly back it up (covered in snapshot chapter)
- Ensure correct permissions (`vault:vault` or `100:100`)

```
sudo chown -R 100:100 /opt/vault-prod/data
```

Step 4: TLS Certificates

Vault **requires TLS** in production to ensure secure communication between clients and the Vault server.

You can:

- Use Let's Encrypt (with a public domain)
- Use **self-signed certificates** for internal deployments or IP-based access

Create a folder to store your certs:

```
mkdir -p /opt/vault-prod/config/certs
```

Generating Self-Signed TLS Certificates

You can create a self-signed TLS certificate using OpenSSL for:

- Internal FQDN (e.g., `vault.internal.domain`)
- Static IP address (e.g., `192.168.1.10`)

Create a file named `vault.cnf`:

```
[req]
default_bits       = 2048
prompt             = no
default_md         = sha256
distinguished_name = dn
req_extensions     = req_ext
x509_extensions    = v3_ca

[dn]
C  = US
ST = CA
L  = San Francisco
O  = Vault Self-Signed
OU = Vault Dev
CN = vault.internal.domain

[req_ext]
subjectAltName = @alt_names

[v3_ca]
subjectAltName = @alt_names
basicConstraints = critical,CA:TRUE
keyUsage = critical, digitalSignature,
keyEncipherment
extendedKeyUsage = serverAuth
[alt_names]
DNS.1 = vault.internal.domain
```

```
IP.1 = 192.168.1.10
Now generate the key and certificate:
openssl req -x509 -nodes -newkey rsa:2048 \
  -keyout /opt/vault-prod/config/certs/vault.key \
  -out /opt/vault-prod/config/certs/vault.crt \
  -days 365 \
  -config vault.cnf
```

This command:

- Generates a 2048-bit RSA private key
- Issues a self-signed certificate valid for 1 year
- Supports both the DNS name `vault.internal.domain` and IP `192.168.1.10`

> Replace the CN and SANs (`alt_names`) with values matching your deployment.

Working with IP vs Domain Name

Scenario	`api_addr`	TLS Cert CN	Use Case
IP-based setup	`https://192.168.1.10:8200`	CN = IP + SAN	Internal, dev, no DNS
Domain-based setup	`https://vault.internal.domain:8200`	CN = FQDN	Internal with DNS or hosts file

Make sure:

- The `CN` or SAN matches the `api_addr`
- Vault clients connect using the same address configured in `api_addr`

Step 5: Start Vault with Docker

Here's the command:

```
docker run -d --name vault \
  --cap-add=IPC_LOCK \
  -p 8200:8200 \
  -p 8201:8201 \
  -v /opt/vault-prod/config:/vault/config \
  -v /opt/vault-prod/data:/vault/data \
  -v /opt/vault-prod/config/certs:/vault/config/certs
\
  -e VAULT_ADDR='https://0.0.0.0:8200' \
  --restart=always \
  hashicorp/vault:latest server
```

`--cap-add=IPC_LOCK` enables memory locking to prevent secrets from being swapped to disk.

Step 6: Initialize and Unseal

Once running, use a separate terminal to exec into the Vault container:

```
docker exec -it vault vault operator init
```

You'll get:

- 5 unseal keys
- 1 root token

Save these **securely**.

Then unseal Vault:

```
docker exec -it vault vault operator unseal <unseal-key>
```

Repeat 3 times.

Final Validation

If everything went well, visiting:

```
https://<<YOUR_IP_OR_FQDN_HERE>>:8200
```

Should show you the Vault UI.

Security Best Practices

- Never run Vault without TLS
- Remove the root token after bootstrap
- Enforce ACL policies and AppRoles
- Use audit devices (see audit devices chapter)
- Snapshot regularly and secure backups
- Mount data volumes with `noexec`, `nosuid`

Recap

Running Vault in production with Docker gives you containerized control, fast iteration, and repeatability but it's still a production-grade security platform. In this chapter, you learned how to:

- Prepare a secure Vault deployment using Docker
- Configure persistent Raft storage
- Secure your deployment with TLS

- Understand `mlock`, data volumes, and IPC
- Handle IP vs domain-based TLS configs
- Initialize, unseal, and talk to Vault

Chapter 26: Production Hardening Guide

Vault is designed to be secure by default, but in production, defaults alone won't save you. Hardening your Vault deployment is not just about compliance, it's about protecting the most sensitive data in your entire infrastructure.

This chapter covers everything you need to **lock down Vault in production**, including:

- Network security
- TLS and certificates
- Authentication controls
- Audit logging
- OS and file system hardening
- Token and lease management
- Rate limiting
- Recommended configuration flags

1. Use TLS Everywhere

Vault **requires** TLS in production. This protects the Vault API from eavesdropping, MITM attacks, and credential theft.

Generate and Use Strong Certificates

- Use an **internal CA** or **Vault's PKI engine** to generate trusted certs.
- Avoid self-signed certificates unless Vault is deployed in complete isolation.

Example (via OpenSSL):

```
openssl req -x509 -nodes -days 365 \
  -newkey rsa:4096 \
  -keyout vault.key \
  -out vault.crt \
  -subj "/CN=internal.vault.local"
```

Vault Configuration for TLS

```
listener "tcp" {
  address       = "0.0.0.0:8200"
  tls_cert_file = "/etc/vault/tls/vault.crt"
  tls_key_file  = "/etc/vault/tls/vault.key"
}
```

2. Restrict Network Access

Vault should **never be exposed directly to the public internet**.

Limit Access via Firewalls or Security Groups

Allow access to Vault only from:

- Internal services that authenticate to Vault
- Admin networks or bastion hosts

Recommended ports:

Port	Description
8200	Vault API / UI
8201	Cluster communication (Raft)

Block 8200 from all external IPs.

3. Run Vault as a Non-Root User

Vault does not need root privileges to function.

Create a vault user:

```
useradd --system --home /etc/vault.d --shell
/bin/false vault
```

Set ownership:

```
chown -R vault:vault /etc/vault.d /opt/vault
```

Update your `vault.service`:

```
[Service]
User=vault
Group=vault
```

4. Enable Memory Locking (`mlock`)

This prevents secrets from being written to disk (via swap).

```
[Service]
LimitMEMLOCK=infinity
```

Also give the Vault binary permission to lock memory:

```
setcap cap_ipc_lock=+ep /usr/local/bin/vault
```

5. Disable the Root Token

The root token should **not be used** beyond initial bootstrapping. Revoke it and rely on tightly scoped policies and real authentication methods.

Use Role-Based Auth Methods

Recommended production methods:

- `userpass` or `LDAP` for admins
- `AppRole`, `Kubernetes`, or `TLS` for apps

6. Enable Audit Logging

Audit logs are the only way to **track who did what in Vault**.

Configure File-Based Audit Log

```
vault audit enable file
file_path=/var/log/vault_audit.log
```

Ensure file is owned by `vault` and not world-readable.

7. Configure Token and Lease Security

Set tight defaults for:

```
# vault.hcl
default_lease_ttl = "1h"
```

```
max_lease_ttl     = "24h"
```

Set these in your auth methods as well (e.g., AppRole):

```
vault write auth/approle/role/my-app \
  token_ttl=60m \
  token_max_ttl=2h \
  secret_id_ttl=30m
```

8. Enable Rate Limiting

Vault can be overwhelmed if a client loops or is misconfigured. Rate limiting protects Vault from unintentional DoS.

9. Use Integrated Storage with Raft

Avoid file-based or dev backends in production.

Use:

```
storage "raft" {
  path    = "/opt/vault/data"
  node_id = "vault-1"
}
```

- Highly available
- Supports snapshots
- Secure and robust

10. Harden the Vault via OS

- Use **Ubuntu LTS** or **RHEL Minimal**
- Disable unnecessary services (`systemctl disable`)

- Remove compilers, shells, user tools
- Use `iptables` or `ufw` to lock ports
- Apply automatic security updates

11. Automate Backups & Snapshots

Set up regular Vault snapshots:

```
vault operator raft snapshot save /backups/vault-
$(date +%F).snap
```

Use `cron` and back them up securely.

12. Enforce Configuration Checks

Vault's config is **plaintext**; treat it as sensitive.

Use automated scanners to detect issues:

- Ownership
- Permissions
- TLS enabled
- Token TTL

Recap

Area	Recommendation
TLS	Always enabled, use valid certs
Network	Internal only, restrict 8200/8201
Audit Logs	File-based, protected

Area	Recommendation
Auth Methods	Root token disabled, roles used
OS User	Vault runs as non-root
`mlock`	Enabled with systemd & `setcap`
Storage	Raft, secured, regularly backed up
Rate Limiting	Via proxies or service mesh
Snapshots	Scheduled, versioned backups

Trust, but verify. Every auth method, secret engine, and configuration should be revisited periodically. Secrets don't expire—**you have to rotate them**.

Chapter 27: Creating PGP Keys for Vault Security

Vault supports advanced cryptographic operations and secure workflows, especially for unsealing, root token recovery, and secure key distribution. One of the mechanisms Vault supports in these workflows is **PGP (Pretty Good Privacy) encryption**. PGP ensures that sensitive values (like unseal keys or root tokens) are shared securely and can only be decrypted by intended recipients. In this chapter, you'll learn how to generate a secure PGP keypair using the GnuPG (GPG) tool.

Why PGP is Important in Vault Workflows

HashiCorp Vault supports PGP for:

- **Encrypting unseal keys** when using Shamir's key sharing.
- **Securely delivering the root token** during the `vault operator generate-root` process.
- **Secure automation and scripting**, where secrets need to be shared among team members securely.

When you supply a PGP public key, Vault can encrypt sensitive information in a way that only the holder of the corresponding private key can decrypt.

Installing GPG

To work with PGP keys, we use the GPG command-line utility (GNU Privacy Guard).

Step 1: Check if GPG is Installed

```
gpg --version
```

If it is not installed, install it with one of the following:

- **Ubuntu/Debian**:

```
sudo apt update && sudo apt install gnupg
```

- **macOS (using Homebrew)**:

```
brew install gnupg
```

- **For alpine linux:**

```
apk add gnupg
```

Generating a PGP Keypair

Step 2: Create Your PGP Key

Use the following command to launch the interactive key creation wizard:

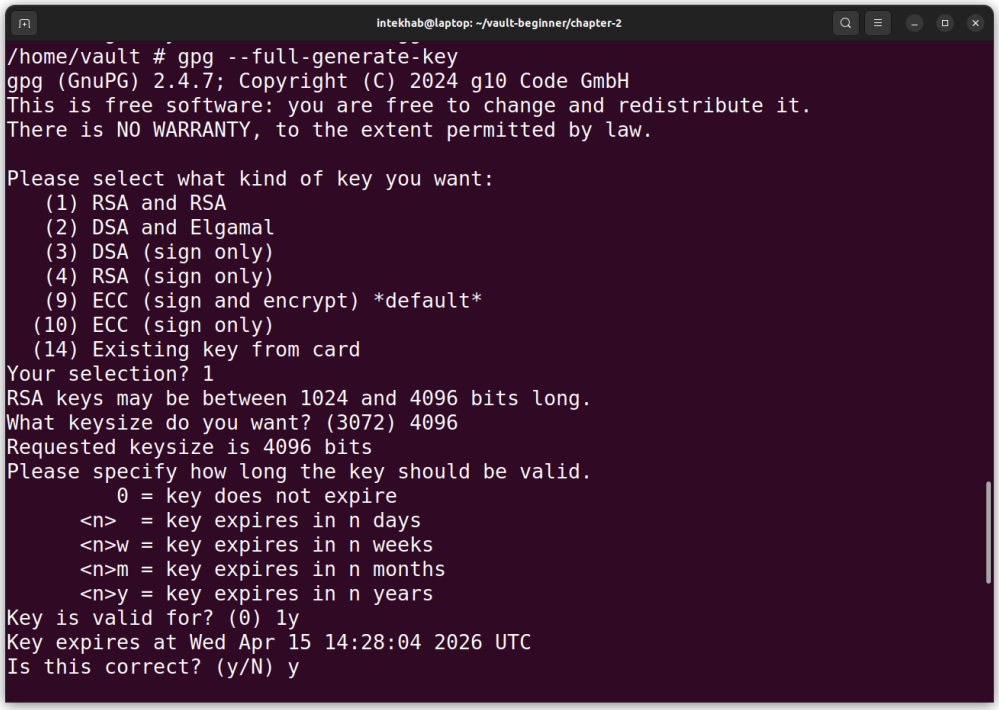
```
gpg --full-generate-key
```

You will be prompted to answer several questions:

1. **Key type** – Choose the default (`1`) for RSA and RSA.
2. **Key size** – Choose at least `4096` bits for strong security.
3. **Expiration** – You can specify a duration (e.g., `1y` for one year) or set it to never expire (`0`).

4. **User ID**:
    - Real name (e.g., `Vault Operator`)
    - Email address (e.g., `vault@example.com`)
5. **Passphrase** – Choose a strong, memorable passphrase to protect the private key.

Example Output



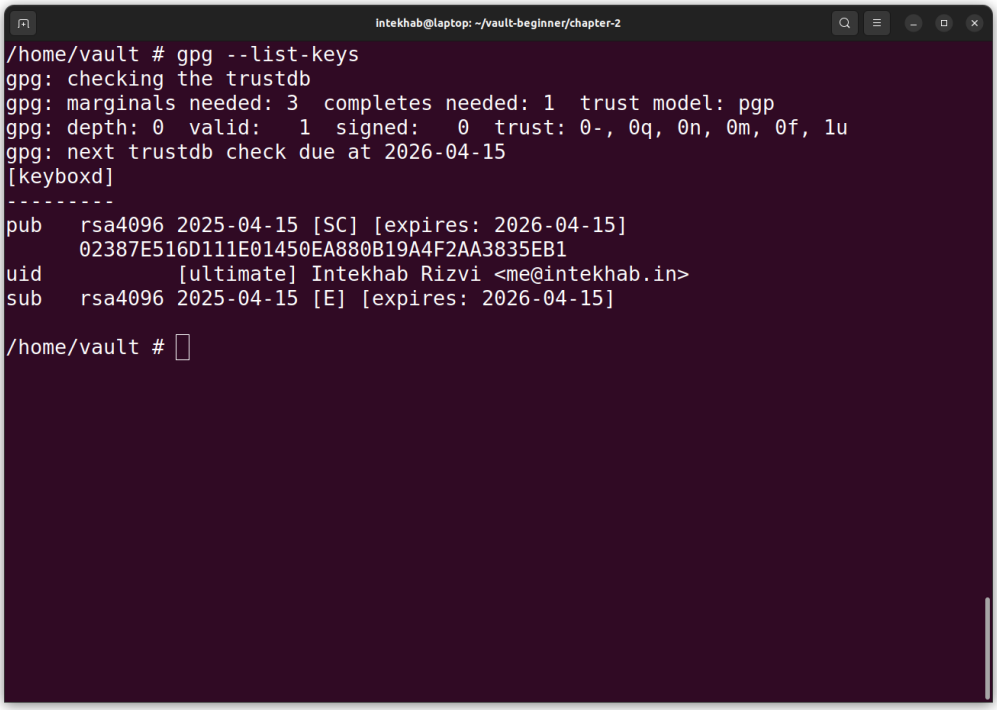
Once the key is created, GPG will display a confirmation and begin creating the keypair.

Listing and Managing PGP Keys

Step 3: View Your Key

```
gpg --list-keys
```

This will output something like:



The long string under `pub` is your key ID (fingerprint).

Exporting the Keys

Step 4: Export Public Key

You'll use this when configuring Vault commands:

```
gpg --armor --export [YOUR_EMAIL_ADDRESS_HERE] >
public-key.asc
```

This creates an ASCII-armored file `public-key.asc` that you can safely share with Vault.

To view the public key directly:

```
gpg --armor --export [YOUR_EMAIL_ADDRESS_HERE]
```

Step 5: Export Private Key (Optional – For Backups Only)

Only export the private key if you need it for secure backup.

```
gpg --armor --export-secret-key
[YOUR_EMAIL_ADDRESS_HERE] > private-key.asc
```

Important: Store the private key securely. It provides access to everything encrypted with the corresponding public key.

Testing the Keypair

You can quickly test that encryption and decryption work:

Encrypt a Sample Message

```
echo "Hello Vault!" | gpg --armor --encrypt --
recipient [YOUR_EMAIL_ADDRESS_HERE] > message.asc
```

Decrypt the Message

```
gpg --decrypt message.asc
```

Summary

- You created a strong RSA-based PGP key using GPG.
- You exported the public key for use in Vault.
- You optionally exported the private key for backup.
- You verified that encryption and decryption work end to end.