ΠΑΝΕΠΙΣΤΗΜΙΟ
ΙΩΑΝΝΙΝΩΝ

UNIVERSITY OF IOANNINA

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Docker-powered PHP Web App: Featuring PostgreSQL, Secrets Management and Custom CMS

Emmanouil Oikonomidis

Supervisor: Tzallas Alexandros, Dean

ATHENS
JANUARY 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΙΩΑΝΝΙΝΩΝ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

PHP Web App με Χρήση Docker: Ενσωμάτωση PostgreSQL, Διαχείριση Μυστικών και Custom CMS

Εμμανουήλ Οικονομίδης

Επιβέπων: Τζάλλας Αλέξανδρος, Κοσμήτορας Σχολής

ΑΘΗΝΑ
ΙΑΝΟΥΑΡΙΟΣ 2025

Εγκρίθηκε από εξεταστική επιτροπή

Άρτα, 25/06/2025

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

1. Επιβλέπων καθηγητής

   Τζάλλας Αλέξανδρος

2. Μέλος επιτροπής

   Γιαννακέας Νικόλαος

3. Μέλος επιτροπής

   Τσούλος Ιωάννης

© Οικονομίδης, Εμμανουήλ, 2025.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Δήλωση μη λογοκλοπής

Δηλώνω υπεύθυνα και γνωρίζοντας τις κυρώσεις του Ν. 2121/1993 περί Πνευματικής Ιδιοκτησίας, ότι η παρούσα μεταπτυχιακή εργασία είναι εξ ολοκλήρου αποτέλεσμα δικής μου ερευνητικής εργασίας, δεν αποτελεί προϊόν αντιγραφής ούτε προέρχεται από ανάθεση σε τρίτους. Όλες οι πηγές που χρησιμοποιήθηκαν (κάθε είδους, μορφής και προέλευσης) για τη συγγραφή της περιλαμβάνονται στη βιβλιογραφία.

Οικονομίδης Εμμανουήλ

Υπογραφή

[illegible]

ABSTRACT

From an initial curiosity on what it would take to self-host a website with no more than a single computer and an internet connection in one's disposal, a spark of interest and in turn lots of research lead to the topic of this Thesis.

Many of the tools shown or used throughout this document, like Docker, were chosen with that initial curiosity in mind, even though in the end I decided not to do a full dive into publishing a website on a public domain. Instead, a more focused approach on how each of these technologies work, and how they can work together, is followed.

We will get to understand how a PHP web app and its PostgreSQL database can be interpreted as services by bundling them up into Docker, as well as look at a way to safely introduce the database's credentials into the web app using Vault, a secrets manager, and RabbitMQ, a message broker. Additionally, a very basic, custom Content Management System (CMS) solution will be explored, useful for managing the web app's content without directly involving the user with the database.

Originally, all of the research was done completely experimentally and without a particular goal in mind. Starting with Docker, then slowly evolving the services it consisted of by going through a lot of trial and error, more and more goals started taking shape. Browsing through countless official documentation pages, often times linked to by StackOverflow posts in its various forums, testing different solutions for each issue that would crop up and discovering new methods to use throughout the project, it became obvious what I wanted the Thesis to be about.

By the end of this document, we should have a fully functional Docker compose stack, with services that comprise all of the technologies already mentioned, and more.

SUBJECT AREA: Software Engineering

KEYWORDS: DevOps, PHP, PostgreSQL, Docker, Secrets Management

ΠΕΡΙΛΗΨΗ

Μια μικρή περιέργεια πάνω στο θέμα του self-hosting μιας ιστοσελίδας χρησιμοποιώντας μόνο έναν ηλεκτρονικό υπολογιστή και μια σύνδεση στο διαδίκτυο, οδήγησε στο ενδιαφέρον και στην έρευνα του θέματος της Πτυχιακής Εργασίας.

Πολλά από τα εργαλεία που παρουσιάζονται ή χρησιμοποιούνται σε αυτήν την εργασία, όπως το Docker, επιλέχθηκαν με αυτήν την αρχική περιέργεια στο επίκεντρο, αν και στο τέλος αποφάσισα να μην αφοσιώσω την εργασία στη δημοσίευση μιας ιστοσελίδας στο διαδίκτυο. Αντί για αυτό, θα ακολουθηθεί μια διαφορετική προσέγγιση, όπου θα συγκεντρωθούμε πάνω στις τεχνολογίες που επιλέχθηκαν, και πως αυτές αλληλεπιδρούν μεταξύ τους.

Θα καταλάβουμε πως ένα PHP web app και μια βάση δεδομένων PostgreSQL μπορούν να ερμηνευτούν ως υπηρεσίες με το Docker, και πως μπορούμε να εισάγουμε τα διαπιστευτήρια της βάσης στο web app με ασφαλή τρόπο, χρησιμοποιώντας το Vault, ένα διαχειριστή μυστικών, και το RabbitMQ, έναν message broker. Επίσης θα δούμε μια custom υλοποίηση ενός βασικού Content Management System (CMS), χρήσιμο προς το χρήστη για τη διαχείριση περιεχομένου χωρίς αλληλεπίδραση με τη βάση δεδομένων.

Στην αρχή, όλη η έρευνα έγινε πειραματικά, χωρίς κάποιο συγκεκριμένο στόχο. Αρχίζοντας με το Docker, και αναπτύσσοντας τον όγκο των υπηρεσιών του, σιγά σιγά διάφοροι στόχοι πήραν μορφή. Με αρκετή περιήγηση στο διαδίκτυο, και μετά από πολλά επίσημα documentation στα οποία πολλές φορές κατέληξα μέσω διάφορων φόρουμ του StackOverflow, έγινε τελικά προφανές το θέμα για το οποίο ήθελα να γράψω την Πτυχιακή Εργασία.

Μέχρι το τέλος αυτού του εγγράφου, θα έχουμε έναν ολοκληρωμένο, λειτουργικό «σωρό» υπηρεσιών στο Docker, που θα περιλαμβάνει όλες τις τεχνολογίες που αναφέρθηκαν, και ακόμη περισσότερες.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μηχανική Λογισμικού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: π.χ. DevOps, PHP, PostgreSQL, Docker, Διαχείριση Μυστικών

Contents

TABLE OF LISTINGS

TABLE OF FIGURES

LIST OF ABBREVIATIONS

CMS	Content Management System
WSL	Windows Subsystem for Linux
YAML	Yet Another Markup Language
HTTP	HyperText Transfer Protocol
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
PGP	Pretty Good Privacy
CLI	Command Line Interface
API	Application Programming Interface
cURL	client URL
HCL	HashiCorp Configuration Language
TTL	Time-To-Live
RPC	Remote Procedure Call
PDO	PHP Data Objects
AMQP	Advanced Message Queuing Protocol
CSRF	Cross-Site Request Forgery
SEO	Search Engine Optimization

It’s quite an interesting thing, the amount of different ways there are to do a particular task when what one is looking for is to do that task all by themselves. That was the case with researching the topic of this Thesis, but was also true of the entire reason I picked it to begin with. The large variety of options to go through meant that I could go with the one that is most efficient for my needs, but also the one that is the most fun to implement.

What I was curious about at first, was how would one go about publishing a website to a domain, assuming they had already made one? The many more questions that arose from just that one curiosity eventually led to learning about self-hosting, and discovering a powerful containerization tool called Docker. For what it’s worth, the Thesis itself won’t be going into all the intricacies of what it takes to fully prepare a project to be published online, however the jump from where the analysis of all the elements of this Thesis stops and where the research of actually forwarding a website to the public starts, is rather small.

The main goal to be accomplished in this article is to leverage Docker and configure it to act as the server of a multi-service stack that encompasses a website, database, secrets manager, admin page (CMS) that will manage the website’s content, and the backend that connects all of those services together with RabbitMQ. To be able to see everything working together in action, at least a basic understanding of how Docker operates is required, and that’s what the first chapter will cover. After that, another chapter will be dedicated to fetching secrets with HashiCorp’s Vault, and the reason why it’s beneficial to even consider secrets in the setup. Next, an example build based on HTML, CSS & Javascript will be introduced to the Docker environment, which includes the process of connecting to a database using PHP Data Objects (PDO), tying up the static build with the secret fetching backend from chapter 2. One of the most important parts of the Thesis is explored in the 4th chapter, where an admin page is created with the purpose of managing content in the main build by accessing and executing database queries bundled up in functions for repeatable calls. Of course, by creating more files we are essentially adding more weak points and expanding the attack surface of the server at the same time, and is partly why the last chapter was included, as it describes some safeguards and best practices on protecting sensitive files that we wouldn’t want attackers accessing, or executing maliciously.

1 Introduction to Docker

The idea of creating a server that would host an entire suite of services, including the website itself, came from my general interest in how does one even begin with hosting a website if they had one ready to be published. I had learned a lot about writing code that can make pages look cool and do useful things in my internship two semesters ago, but I always wondered how the smart people at this company ran the show behind the scenes. Perhaps what I have ended up with isn't exactly their way, but it seems that I have found a good balance for someone looking to start somewhere with little experience, and at the same time being a solution for people that like fiddling with options until they get exactly what they want out of the software.

Something particularly important for the host seems to be their ability to protect the system that everything is running on, and that makes sense. If you want people to access your website, you are subscribing to the idea that they will need to access the system that the website will be hosted on. In my case, I wanted that system to be my personal computer, but not without first taking the right steps to ensure that I wouldn't harm my files or other sensitive information as a result of being too curious. It turns out that the perfect way to do that was to run that server in an isolated space on my computer. Docker was consistently suggested as the go-to software for this purpose: it is to isolate slices or parts of the system by using shared global resources like RAM, processing power, & disc space assigned to unique namespaces (see Shital Shah, 2016, for more detail), to act as isolated environments for applications based on whatever the user decides to run inside. Apparently, Docker can be much more.

1.1 Containerization

Having already talked about isolating an instance of something the user desires, it's time to put a name behind that particular process for our intended scope, and specify what the "something" is. The former, is simpler to do than the latter: Containerization. Docker's most basic ability, and perhaps the function we'll be taking the biggest advantage of, is to bundle up code, images, configuration options, and other files with the purpose of skipping the part where the user has to worry about what Operating System they are using Docker on, what their network looks like, and all of the other things that have nothing to do with the bundle that is being containerized. Obviously, that convenience is an added benefit to the original purpose of running an isolated instance of something, but these two go hand-in-hand.

To answer the question of what that "something" is that we can containerize, let's assume one basic example before moving onto a complicated scenario:

Perhaps we are interested in hosting a single HTML file, and a CSS file to go along with it. There are many ways to containerize this basic arrangement, however at the very least we will need an HTTP server, and an Operating System to run that server on.

The HTML markup (shown in Listing 1.1.1) will be printing a simple message on the screen, which should be enough to make sure everything is working properly:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
    <title>My Cool Stack's Index Page</title>
    <link href="/cute.css" rel="stylesheet">
</head>
<body>
    <div class="hello-message">Hello World!</div>
</body>
</html>
```

Listing 1.1.1: HTML code (index.html)

Styling it isn't important, however it will allow us to understand the logic on the freedom we have with containerizing, and so a CSS file will also be included, shown in Listing 1.1.2:

```
html {
    height: 100%;
}

body {
    justify-content: center;
    background: linear-gradient(180deg,
    #456FD0 0%, #3C84D5 69.5%, #3299D9 100%);
}

.hello-message {
    margin: auto;
    left: 0;
    right: 0;
    position: absolute;
    color: #ffffff;
    font-size: 45px;
    top: 40%;
    text-align: center;
}
```

Listing 1.1.2: CSS code (cute.css)

For anything Docker related, it is important to note that all configuration for this Thesis was done with Docker Desktop on a Windows 10 Pro (Version 10.019045, Build 19045) computer, utilizing WSL2 (Windows Subsystem for Linux 2). While that information doesn't matter for the containerization, it would be amiss to omit it as I believe it provides the right context to the process I'm following to get the results that will be presented.

WSL2 is the Linux kernel solution that's powering Docker Desktop on Windows, and upon installing it to the system (if not already available), developers "can leverage Linux workspaces and avoid maintaining both Linux and Windows build scripts" (Docker, n.d.-b).

Two extra files will be used for containerization:

- Compose file
- Dockerfile

First, our compose file will include the configuration of the container, or in other words its settings. This file uses the YAML format, which is a data serialization language that features the useful key-value pair system which is ample for configuration files. Having a folder named “app”, and our HTML & CSS files included in it, we will consider that the build context for the container. In Listing 1.1.3 we can see the compose file config:

```yaml
# Compose file, docker 4.36.0
services:
  server:
    build:
      context: ./app
    ports:
      - 2025:80
```

Listing 1.1.3: Compose configuration (compose.yaml)

YAML is almost reading out the instructions as if it was meant for a human to understand, but some translation will be provided for the reader:

“`server`” is a **service**, which for now is the only service we will define.

The ports that will be exposed read out in this format:

HOST_PORT:CONTAINER_PORT

With the above example then (from Listing 1.1.3), we are choosing to listen to port 2025 on the host machine, and to 80 on the container itself. The distinction between the two is important and will become clearer when we have more services, but for now remember that the left port (host) is what we will access index with, and the right port (container) is what Docker will use internally to forward the page as it needs to. Note that port 80 will deliberately not be getting used on the host machine, because we are not aiming on publicizing the website at this stage, which is what exposing port 80 on the host’s side would do (assuming your router/modem is also exposing that port by default). The same goes for any other ports that are suspected to be exposed by default on the router/modem, and so an arbitrary one is picked for development.

The build context, as mentioned before, houses all the files we want to containerize, but just as importantly, the Dockerfile:

```dockerfile
# syntax=docker/dockerfile:1
# Server

FROM nginx:alpine
COPY . /usr/share/nginx/html
```

Listing 1.1.4: Dockerfile

The language in a Dockerfile describes instructions to what Docker should do with the particular service that is associated with the corresponding build context defined in the compose file. It is also possible to have multiple Dockerfiles so that each service build can be orchestrated with its own settings. Note that not every service needs a Dockerfile, and many services that will be discussed in later chapters will require a slightly different setup involving mainly the compose file.

In Listing 1.1.4, we can see that our Dockerfile has two instructions: `FROM`, and `COPY`. The first one, says that an image called "`nginx`" will be used to build our container, and is called a base image since we will be putting all of our files on top of it. Additionally, from the "`nginx`" image, the version "`alpine`" will be used. This particular image is pretty lightweight, and is perfect for this example with only an HTML & CSS file involved.

The `COPY` instruction simply copies whatever files exist in the build context (.) to the destination in the container defined right after: "`/usr/share/nginx/html`". This specific folder will be picked to move our files to, because it's where nginx's root folder is configured to be at by default, and it's where it will be looking for index. As already mentioned, any files within the build context will be copied over at this stage, so if there are any sensitive files or even entire folders that we wish were ignored, we could introduce another file to the context folder named ".dockerignore":

```
**/Dockerfile*
**/.dockerignore
**/compose.y*ml
```

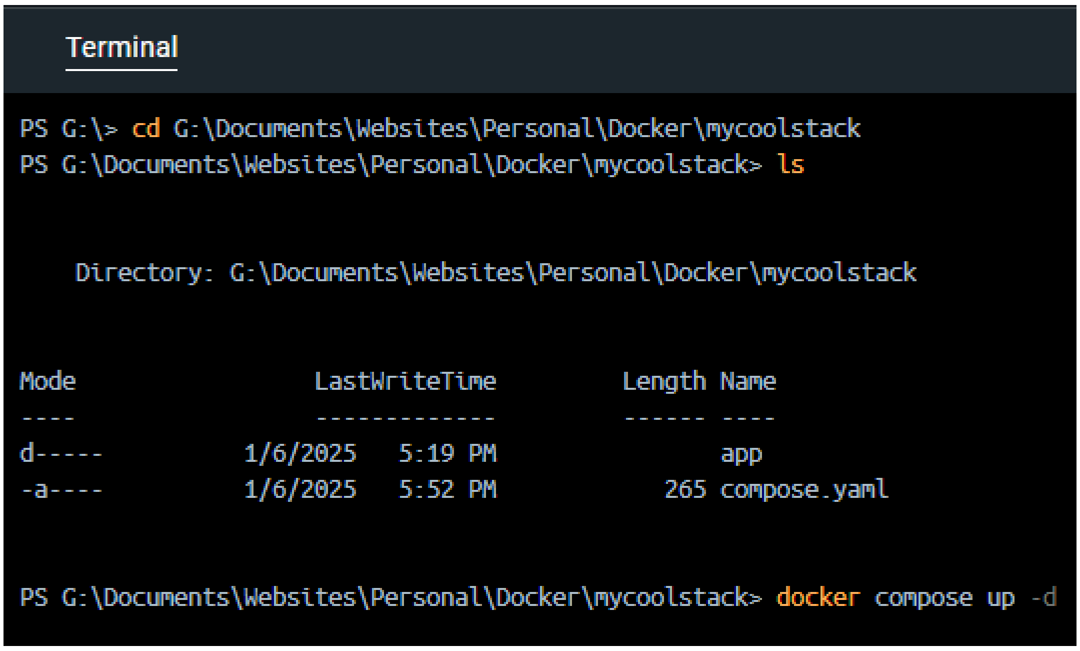
Listing 1.1.5: .dockerignore examples

Our files are now ready to be containerized! The folder structure on the host side that would be valid for the Dockerfile & compose.yaml settings from Listing 1.1.3 & 1.1.4, would look like this:

```
mycoolstack/
├── app/
|       ├── index.html
|       ├── cute.css
|       ├── .dockerignore
|       └── Dockerfile
└── compose.yaml
```

Listing 1.1.6: Build context structure

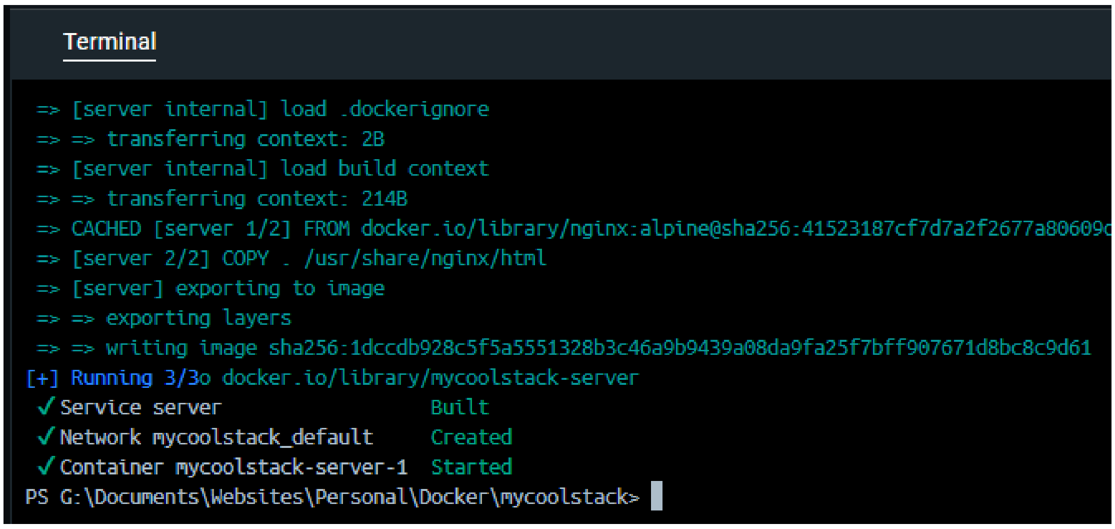
There’s nothing left but to compose with the settings we chose. In Docker, this can be done using the terminal, by navigating to the root folder of the project on the host with `cd` and finally using “`docker compose up -d`”. This is shown in Figure 1.1.1:



```
PS G:\> cd G:\Documents\Websites\Personal\Docker\mycoolstack
PS G:\Documents\Websites\Personal\Docker\mycoolstack> ls


    Directory: G:\Documents\Websites\Personal\Docker\mycoolstack


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          1/6/2025   5:19 PM                app
-a----          1/6/2025   5:52 PM            265 compose.yaml


PS G:\Documents\Websites\Personal\Docker\mycoolstack> docker compose up -d
```

Figure 1.1.1: Terminal commands

If everything is syntactically correct in both the Dockerfile and the compose.yaml file, Docker will finish composing by creating, then starting the container, as shown in Figure 1.1.2:



```
 => [server internal] load .dockerignore
 => => transferring context: 2B
 => [server internal] load build context
 => => transferring context: 214B
 => CACHED [server 1/2] FROM docker.io/library/nginx:alpine@sha256:41523187cf7d7a2f2677a80609c
 => [server 2/2] COPY . /usr/share/nginx/html
 => [server] exporting to image
 => => exporting layers
 => => writing image sha256:1dccdb928c5f5a5551328b3c46a9b9439a08da9fa25f7bff907671d8bc8c9d61
[+] Running 3/3o docker.io/library/mycoolstack-server
 ✓Service server                Built
 ✓Network mycoolstack_default   Created
 ✓Container mycoolstack-server-1  Started
PS G:\Documents\Websites\Personal\Docker\mycoolstack> 
```

Figure 1.1.2: Compose results

What “starting the container” can possibly mean, varies from build to build. In our current build with nginx as the base image, the directive for the service is for nginx to not self-daemonize, that is to not run in the background but in the foreground instead:

```
CMD ["nginx", "-g", "daemon off;"]
```

This particular instruction can be found in the Dockerfile that builds `nginx:alpine` itself, because after all when we use `FROM nginx:alpine` we are also executing all of the instructions of that image's Dockerfile as well as the ones in our own Dockerfile. The reason why we need nginx to be running in the foreground in our Docker container, is because containers need a single foreground process to be running, otherwise they are considered inactive and halt. This design is deliberate, and it is so because **each one container is supposed to be one service**, allowing for dependencies between services to be straightforward, where if for any reason a container stops working, then any number of other services that depend on it halt and wait for it to become available again. That simply wouldn't be possible if each container had more than one foreground process at a time, and any failed background processes wouldn't shut down the container, meaning they'd be dead without us knowing, potentially causing issues.

In Figure 1.1.2, we also see "Network mycoolstack_default Created". This is the internal network used by Docker to route containers together, and listen for traffic on any ports we happened to expose in the compose stage.

To see if the index file was found by the nginx process, we can navigate to `localhost:2025` in a browser:

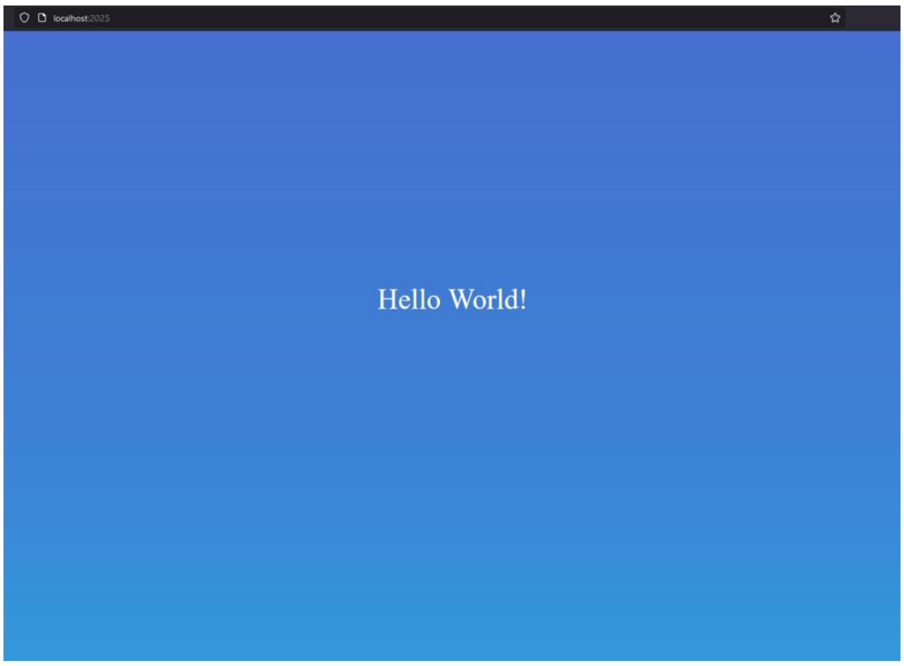


Figure 1.1.3 - Container index

Getting the “Hello World” screen just about confirms everything is working as expected, however we can still take a look under the hood.

If we browse our container’s files and navigate to the path we copied our HTML & CSS files in, we can see (in Figure 1.1.4) that they were correctly placed in `/usr/share/nginx/html`. The rest of the files in this container were automatically created using the `FROM nginx:alpine` instruction in the compose file:

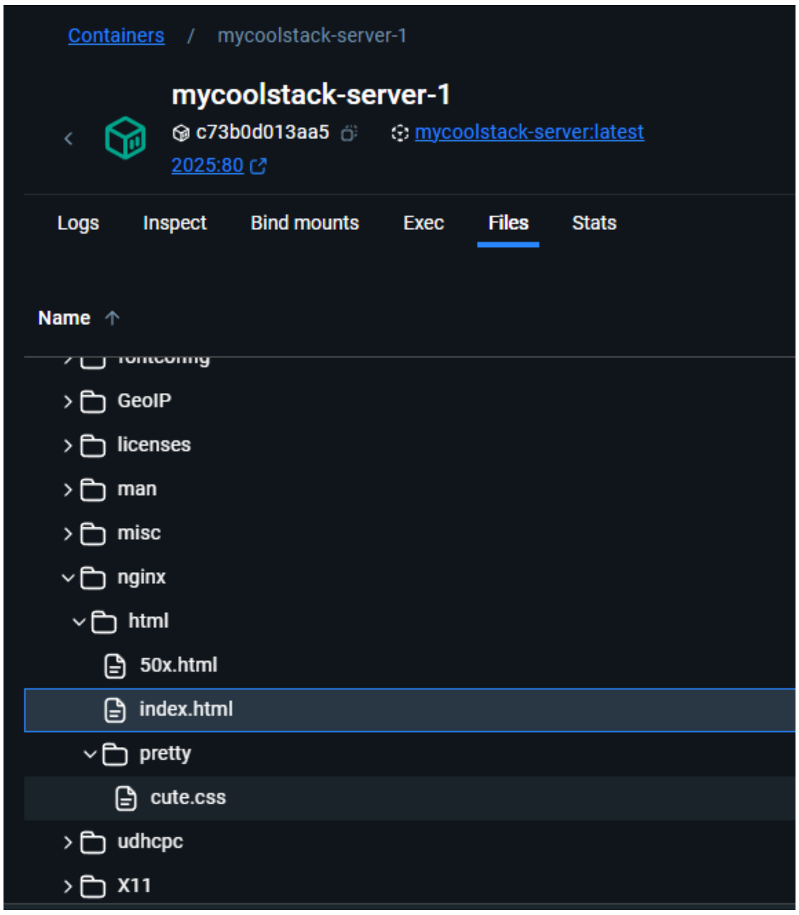


Figure 1.1.4 - Container file browser

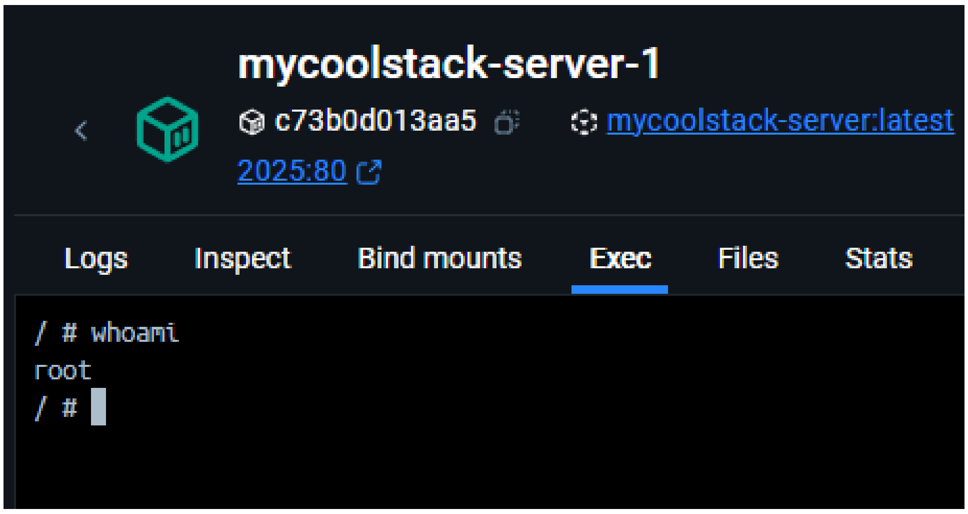
This container is just under 55MBs in size, which will generally only get bigger from here as we introduce more utilities in each container, while others will come as standalone images using entirely their own settings and file structures needing no additional files. To make the distinction clear between an image and a container, let’s say that a container is an image that we have composed, and so technically even `nginx:alpine` is an image, until we composed it to build a container.

1.2 Vulnerabilities & Permissions

Having created our container then confirmed that our page is reachable by invoking the exposed port at the `localhost:2025` address, the next step is to ensure that our container, however simple or basic, is fulfilling a checklist of best practices in order to mitigate vulnerabilities against a supposed attack to our Docker environment. Note that "mitigate" is not chosen by chance to describe the best we can do to protect our system, and that is because there is no recipe that provides 100% invulnerability against attacks, and each systems administrator will always have to compromise utility for security with discretion, and to some degree it really just comes down to "This is good enough.".

There are steps that we can take that can have a massive impact on security. This subchapter will specifically focus on those impactful actions, by using some of them as principles to the container we created in subchapter 1.1.

In our original Dockerfile instruction set I omitted an important step so that I can properly asses it in its own section, and that is to switch the container user to one with lowered privileges. Essentially, since we are using the `alpine` version of `nginx` that means that we have an Operating System, and as is tradition, a user needs to be created for that Operating System. Another useful feature of Docker is the ability to execute commands on that Operating System through the container's exec console:



```
/ # whoami
root
/ #
```

Figure 1.2.1 - Container exec shell

As shown in Figure 1.2.1, the default assigned user if the instruction is omitted is the root user, and while that may not seem like an issue at first, remember that the container's purpose is to provide a particular service and nothing more. When root privileges are assigned to a container, what this means is that it won't simply be able to provide the service we had planned for it, it will also be able to operate outside of that scope if only it was instructed to by a bad actor, i.e. an attacker. For this reason, we will always opt for switching to a user with non-privileged access wherever applicable, and also restricting permissions to files we introduce to the container if these are going to be performing important tasks, such as writing into other files, or containing server logic.

To take the right actions, we can look at the official nginx image overview on hub.docker.com/nginx, where it is described that in order to run nginx as a non-root user we'll need to change the `nginx.conf` file, which is located in `/etc/nginx/nginx.conf`. Specifically, in that file we will replace the current `pid` with `/tmp/nginx.pid` and enable temp paths by modifying the http context with the lines shown below, in Listing 1.2.1:

```
client_body_temp_path /tmp/client_temp;
proxy_temp_path       /tmp/proxy_temp_path;
fastcgi_temp_path     /tmp/fastcgi_temp;
uwsgi_temp_path       /tmp/uwsgi_temp;
scgi_temp_path        /tmp/scgi_temp;
```

Listing 1.2.1: nginx.conf changes

These changes can be made on a copy of that file on our host machine, by choosing to save it first. All files in the container can be saved locally, by going into the "Files" tab as shown in Figure 1.2.2:

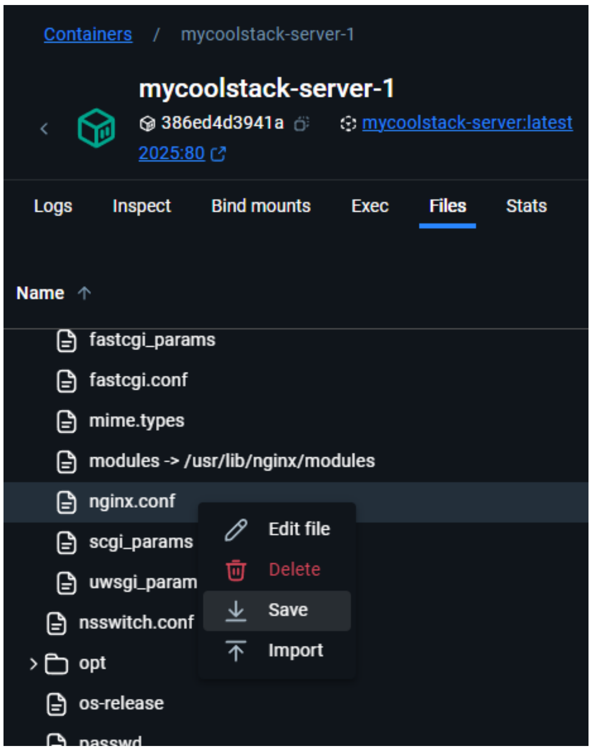


Figure 1.2.2 - Saving a file in Docker

The reason we won't be editing the configuration file inside the container, is because we'd rather we made the user switching process portable, that is that we can simply import all of that logic in our Dockerfile and use those settings to create more containers in the future if needed. If instead we chose to make these changes inside of the container at runtime, we are not only making it harder to deploy a container like that in the future, but we will inevitably run into issues with our changes not persisting long-term. One way to understand this, is that since the container was created using a set of instructions (both defined in our Dockerfile and that of nginx), if we were to ever introduce more files to the container and thus needing to build the container again, we would run that set of instructions again, overwriting all of the configuration that was done manually until that point. Therefore, it is beneficial to stick to containerizing images with portability in mind. Docker (n.d.-a), embraces the idea of building ephemeral containers, "meaning that the container can be stopped and destroyed, then rebuilt and replaced with an absolute minimum set up and configuration."

Now that we have `nginx.conf` configured and ready to be imported, we can adjust our Dockerfile to 1) include it in the right directory, and 2) switch user right before starting the container.

Currently, we are copying all of the files that are inside the build context straight into `/usr/share/nginx/html` which is correct for our HTML & CSS files, however it's the wrong directory for `nginx.conf`, so some restructuring will be necessary on the host files. Personally, I prefer mirroring the structure that the container has, inside of my own build context, that way I can simply use `COPY . .` which copies everything from the build context into the root directory of the container. The folder structure in the build context would look like what's shown below, in Listing 1.2.2:

```
mycoolstack/
├── app/
|     ├── usr/
|     |     └── share/
|     |           └── nginx/
|     |                 └── html/
|     |                       ├── index.html
|     |                       └── cute.css
|     ├── etc/
|     |     └── nginx/
|     |           └── nginx.conf
|     ├── .dockerignore
|     └── Dockerfile
└── compose.yaml
```

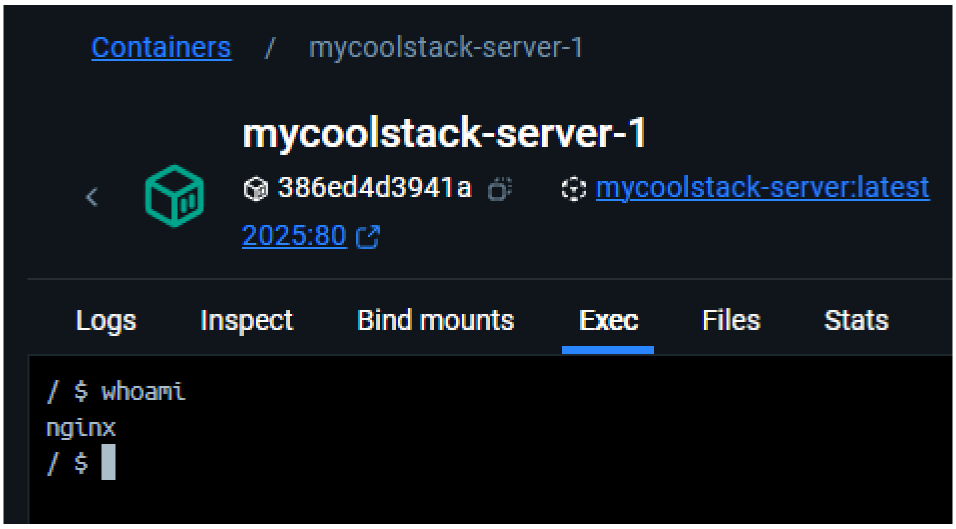
Listing 1.2.2: Build context restructured

Composing the image with this folder structure in the context should direct all of the files to their respective place in the container.
Now, the Dockerfile needs to be revised to copy the files based on the new folder structure, and for the user to switch at the end. This is shown in Listing 1.2.3, as follows:

```
# syntax=docker/dockerfile:1
# Server

FROM nginx:alpine
COPY . .
USER nginx
```

Listing 1.2.3: Revised Dockerfile

Now we can build the container again, and run the same `whoami` command in the shell:



```
/ $ whoami
nginx
/ $
```

Figure 1.2.3 - Container exec shell

The distinction between what user is assigned to us in the container is also obvious when we compare the shell symbol from Figure 1.2.1, the `#` symbol which denotes a root user, and the `$` symbol in Figure 1.2.3 which is for non-root users.

Since we now have successfully lowered our privileges in the container, the next goal is to also protect the files we will be introducing to the container from tampering. By default, any file we copy over in the compose stage will get `chmod 755` for its permission mode. That is, **read + write + execute** for the owner of the file, and **read + execute** for the group it belongs to and other users. The idea like before, is to set the permissions to the most minimal mode we can. If we are not utilizing groups in our Docker container (for example working with other developers under the same group), then the only important triplet value to set is what the owner can do with that file, unless other users need access to it too, like a second service. The "more than one service per container" idea should be entertained extremely conservatively, because like it was mentioned before, is not a best practice.

When introducing files to the container, we should also ensure that the ownership of those files belongs to us, or rather the user we want to run the container as. By doing this, we are making sure that when the permission mode is applied, the access level we are granting to the files is clear, and that none of the files will belong to root, locking the non-root user from access.

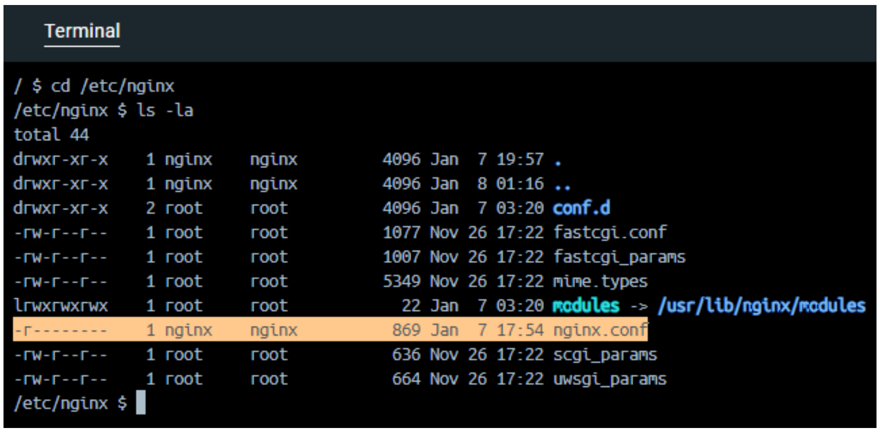
For files that are meant to be executed, like a `.sh` file we need both **read + execute** mode, while markup files like HTML or a scripting language like PHP can do with just the **read** mode. Ultimately, by playing around with different permission modes for each file we can figure out what the most minimal one is and change the settings accordingly. The most important thing of note, is that we shouldn't allow the **write** mode, unless the file we are introducing to the container is supposed to be written into from another file.

```
# syntax=docker/dockerfile:1
# Server

FROM nginx:alpine
COPY --chown=nginx:nginx . .
RUN chmod 0400 /etc/nginx/nginx.conf
RUN chmod 0400 /usr/share/nginx/html/index.html
RUN chmod 0400 /usr/share/nginx/html/cute.css
USER nginx
```

Listing 1.2.4: Changing ownership, and adding permissions in Dockerfile

As shown in Listing 1.2.4, with `COPY --chown=nginx:nginx . .` we are essentially carrying out two operations in one line: Copying all the files from the build context to the root directory in the container, and changing the ownership to user nginx and user group nginx (group is not important, just illustrated here to show it's possible). Then, each file gets the most minimal permission possible, which in this case happened to be just **read** mode for all of them.

By navigating to each directory with the container shell, we can see that both the ownership was enforced and the permissions were successfully altered:



```
/ $ cd /etc/nginx
/etc/nginx $ ls -la
total 44
drwxr-xr-x    1 nginx    nginx         4096 Jan  7 19:57 .
drwxr-xr-x    1 nginx    nginx         4096 Jan  8 01:16 ..
drwxr-xr-x    2 root     root          4096 Jan  7 03:20 conf.d
-rw-r--r--    1 root     root          1077 Nov 26 17:22 fastcgi.conf
-rw-r--r--    1 root     root          1007 Nov 26 17:22 fastcgi_params
-rw-r--r--    1 root     root          5349 Nov 26 17:22 mime.types
lrwxrwxrwx    1 root     root            22 Jan  7 03:20 modules -> /usr/lib/nginx/modules
-r--------    1 nginx    nginx          869 Jan  7 17:54 nginx.conf
-rw-r--r--    1 root     root           636 Nov 26 17:22 scgi_params
-rw-r--r--    1 root     root           664 Nov 26 17:22 uwsgi_params
/etc/nginx $
```

Figure 1.2.4 - Permissions in the container

The vulnerabilities covered in this subchapter are by no means exhaustive of the long list of possible ways an attacker could grant themselves elevated privileges in our container, or even escape the service and gain host-level access, but a few of them definitely deserve to be mentioned because of the level of impact they can have with only a little bit of effort by the system administrator.

The non-root user directive was discussed at length because it was one of the only vulnerabilities that I wanted to talk about that needs action as opposed to inaction. To switch to a non-privileged user we had to set a value for `USER` in order to protect against that particular vulnerability, therefore we had to act. Sometimes, it is best that we instead make choices around not acting, and that brings up these next three big vulnerabilities:

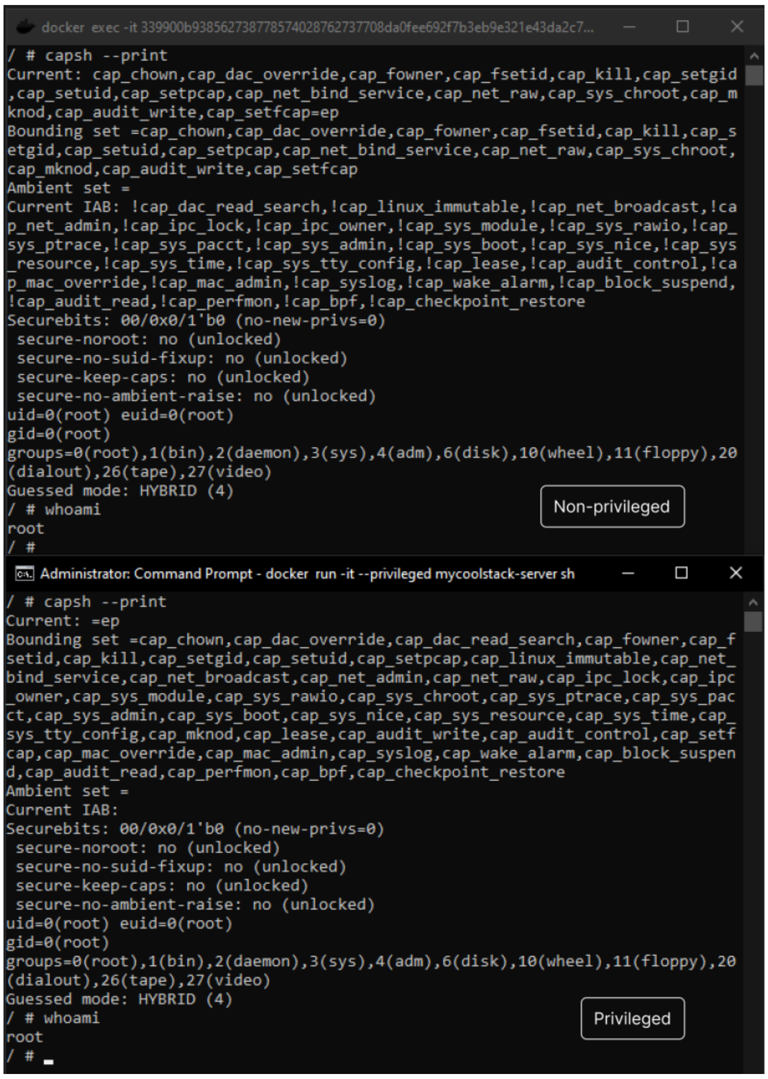
- Running containers with `--privileged`
- Adding `--cap-add=SYS_ADMIN`
- Mounting the Docker Socket

It should be obvious that having great powers in our containers is rather convenient for enforcing necessary changes or options, and that is exactly the reason we opt to lower our privileges at runtime. When performing compose actions, we switch to a non-privileged user at the end of all the operations because most likely they require root access to carry out. It's a solid compromise: we give up our powers after we've done all the necessary work that needs them.

However, Docker allows for certain flags/options when running a build. We composed our container using "`docker compose up -d`" in the last subchapter, where the `-d` flag composed the build in detached mode, that is while not being bound to the standard streams so we can continue using the terminal for other operations. Now, we could run that composed build with different flags depending on our needs, but great care should be taken with any flags that elevate privileges.

Both the flags `--privileged` and `--cap-add=SYS_ADMIN` are designed to give higher than superuser capabilities to the current user, and this can be observed by using `capsh --print` in a normal run Vs. a run with the `--privileged` flag. The next Figure shows the list of capabilities our user has been granted in either case, and it is to be noted that while Docker assigns a default list of capabilities to the user, those can be explicitely dropped with `--cap-drop`. The user is advised to learn about each one and decide which capabilities are best suited for their service, and drop the rest.

The following comparison is made with the user set as root, to highlight just how powerful `--privileged` is.


docker exec -it 339900b938562738778574028762737708da0fee692f7b3eb9e321e43da2c7...
Non-privileged
Administrator: Command Prompt - docker run -it --privileged mycoolstack-server sh
Privileged

```
/ # capsh --print
Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid
,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_m
knod,cap_audit_write,cap_setfcap=ep
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_s
etgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,
cap_mknod,cap_audit_write,cap_setfcap
Ambient set =
Current IAB: !cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcast,!ca
p_net_admin,!cap_ipc_lock,!cap_ipc_owner,!cap_sys_module,!cap_sys_rawio,!cap_
sys_ptrace,!cap_sys_pacct,!cap_sys_admin,!cap_sys_boot,!cap_sys_nice,!cap_sys
_resource,!cap_sys_time,!cap_sys_tty_config,!cap_lease,!cap_audit_control,!ca
p_mac_override,!cap_mac_admin,!cap_syslog,!cap_wake_alarm,!cap_block_suspend,
!cap_audit_read,!cap_perfmon,!cap_bpf,!cap_checkpoint_restore
Securebits: 00/0x0/1'b0 (no-new-privs=0)
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20
(dialout),26(tape),27(video)
Guessed mode: HYBRID (4)
/ # whoami
root
/ #
```

```
/ # capsh --print
Current: =ep
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_f
setid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_
bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc
_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pac
ct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_
sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setf
cap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspen
d,cap_audit_read,cap_perfmon,cap_bpf,cap_checkpoint_restore
Ambient set =
Current IAB:
Securebits: 00/0x0/1'b0 (no-new-privs=0)
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20
(dialout),26(tape),27(video)
Guessed mode: HYBRID (4)
/ # whoami
root
/ #
```

Figure 1.2.5 - Same container, two different run modes

The bounding set shown in Figure 1.2.5 for both non-privileged and privileged runs of the build is the maximum set of capabilities that can possibly be gained by the user, and we can clearly see that in the case of the privileged run we have higher than superuser (root) powers in the container. This discrepancy is because Docker limits even root's capabilities, which can be observed in the non-privileged run in the "current IAB" set, where essentially a bunch of capabilities are dropped (marked with the exclamation mark). By paying closer attention to the capabilities granted with the privileged run, we can see that `cap_sys_admin` was also included in the set of capabilities we can gain.

On top of adding an unreasonable amount of extra capabilities to our user which is in and of itself a dangerous move, a `--privileged` run will also nearly disable all of the protections that come with AppArmor, seccomp, and SELinux. It also allows access to all host devices, by enabling direct access with designated files in the container that interact with hardware/virtual devices on the host that would be inaccessible in the container otherwise.

Whether we add `cap_sys_admin` manually with `--add-cap` or run the container with `--privileged` we are granting the user the ability to mount filesystems within the container. Essentially, what this means is that if our container was compromised by an attacker, it would be easy to for them to do severe damage to our container at best (which we should be able to easily replace), and at worst allow for variable exploits to be used to completely escape the user from the isolated container and perform operations on the host.

Sometimes, Docker images will require `docker.sock` to be mounted as part of the installation. For example, Traefik is a tool that describes itself as "a modern HTTP reverse proxy and load balancer that makes deploying microservices easy", and requires the Docker Socket to be mounted. Another example, is Docker in Docker (DinD) because of course that's a thing, but before we let that thought implode our heads, we can again focus on the fact that this setup too requires the Docker Socket to be mounted.

To mount a file or folder in Docker, is to simply bind a path on the host with a path in the container. In our container for example, a mount could be our HTML file's path `./app/usr/share/nginx/index.html` on the host, binding on `/usr/share/nginx/index.html` on the container. Binding these two paths would allow for every change we make to that file on the host to be instantly reflected on the file on the container, and vise-versa. This is a very useful feature to developers for live changes, but because of how these files inherit permissions, it can also be dangerous.

Mounting the Docker Socket is no exception, only it gets way worse. The Socket is responsible for "providing means to interact with the Docker API directly" (GeeksForGeeks, 2024, para. 15), which spells trouble not only for the container that has mounted the Socket, but the entire environment and even the host. When we give a container access to the tool that directly handles communication with the Docker daemon, which is basically the process that's managing our isolated instance (including any other isolated Docker instance, or network), we are giving that container the ability to perform Docker commands much like the ones we can perform outside of that container to manage it. From there, the only thing missing is a bad actor to orchestrate the proper attack on our system. One of the simple ways this could be done would be with the attacker creating a new container or running an existing container with `--privileged` which meets its own bundle of issues like we discussed earlier.

The short of it, is when we are not sure, we should not mount the Docker Socket, nor run containers with `--privileged` or `--add-cap=SYS_ADMIN` otherwise more damage can be done than good. Of course, all of these principles are to be applied in tandem with running our container with a non-root, unprivileged user, while also being careful with how we manage the permissions in the files we introduce to the container. Much like in many real life scenarios where security is concerned we always describe the process as a multi-step effort and never a one-solves-all solution, and security in containers is no different.

1.3 "Docker, compose up!"

We learned how to set up a basic Dockerfile to containerize our application using a base image in subchapter 1.1, and got to understand a few vulnerability concepts, talked about flags to avoid using with our container, and managing which permissions to give files we introduce to the container in subchapter 1.2.

It's time we moved on to a more advanced setup where we install more utilities, involve multi-container logic, create more services and underline what we want each one to do in the compose stack.

One thing we know, is that we want users to access our application by using a web interface, therefore we want a server to handle HTTP requests. We can also make use of a dynamic way to display data instead of an HTML file, and one way to do that would be with PHP fetching data from a database. While not a requirement, it would be preferred to have a secure way to store the database credentials instead of plaintext in the server, so that could be yet another service. In short, we will need the following services:

1. PHP server
2. Database
3. Secret storage

Since we've already been acquainted with the idea of a live server with the `nginx:alpine` image, we could start expanding on a more complete container that could act as the server in our stack. For this role, we will be using the official PHP image as the base image, and pair it with the Apache tag to specify the server solution we want for PHP.

```yaml
# Compose file, docker 4.36.0
services:
  server:
    build:
      context: ./app/server
    env_file:
      - ./app/server/etc/apache2/envvars
    restart: always
    volumes:
      - ./app/server/var/www/html/index.php:/var/www/html/index.php
    ports:
      - 2025:80 # Public HTTP Port
```

Listing 1.3.1: compose.yaml file

As we will be introducing more and more services in the stack, it's good practice to separate up each service in the host that requires a context or is binding a volume to the container. In Listing 1.3.1, we see that the new location for the server files is bundled up in a folder inside `./app` and the context for the server is now that new folder. Similarly, each new service will correspond to its own folder, with its own Dockerfile (if needed). For the server, a single volume has been bound under `volumes` so we can utilize live changes in our PHP file, but it's not necessary for it to be served by Apache.

Often times we will use certain environment variables defined in an image to set values, right in the compose file. In the above example, we do this with the variable `env_file` which sets the path for Docker to look for to reference envvars values for the server container, but note that these values will be **appended** to the native envvars in the container. Another new element is `restart`, which sets the behavior directive for a container in the case it is forcefully shut down. This value is usually set to `always` if we need our container to try re-launching and is important to the stack because other services are depending on it. The port we will be exposing on the host is again one that is not already exposed by default at the modem/router level, since we don't desire for the server to be publically accessible.

```
# syntax=docker/dockerfile:1
# Server
FROM php:8.2.10-apache
# PostgreSQL + PDO
RUN apt-get update && apt-get install -y libpq-dev \
   && docker-php-ext-configure pgsql -with-pgsql=/usr/local/pgsql \
   && docker-php-ext-install pdo \
   pdo_pgsql \
   pgsql
# Build + Ownership
WORKDIR /
COPY --chown=www-data:www-data . .
# Perms
RUN find /var/www/html/ -type f -exec chmod 0400 {} +
RUN chmod 0404           /etc/apache2/envvars
# Production Configuration: $PHP_INI_DIR = /usr/local/etc/php
RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"
# Switch User
USER www-data
```

Listing 1.3.2: Server's Dockerfile

Using the official PHP image and the Apache tag for our actual server, we will get a head start on installing utilities to make the database connection in the future. The syntax of a multiple `apt-get` that's shown in Listing 1.3.2 will get a lot of use in Dockerfiles, especially because it reduces code space significantly. For permissions, we traverse through all potential files in the `/html` space recursively, applying the minimum mode possible which is just **read** for the owner. The `env_file` we saw in listing 1.3.1 will be getting **read** mode for both the owner and others, because we are neither copying it over nor changing its ownership with `COPY --chown=www-data:www-data . .` and instead will instruct the build to ignore it (with .dockerignore) and let the compose file take care of it on its own, thus it belongs to root. The production configuration for PHP can be used by default with the `RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"` directive, which is a fast and easy way to get started with a commonly used configuration and not have to manually set a bunch of .ini values for PHP. Lastly, we switch to a non-privileged user, leveraging one that is made automatically for us in the base image: `www-data`.

After composing this container, we can again navigate to localhost:2025 and see that our Apache server is properly responding to our HTTP requests. The Logs section in the container shares various useful information about the server, for example the HTTP request itself:

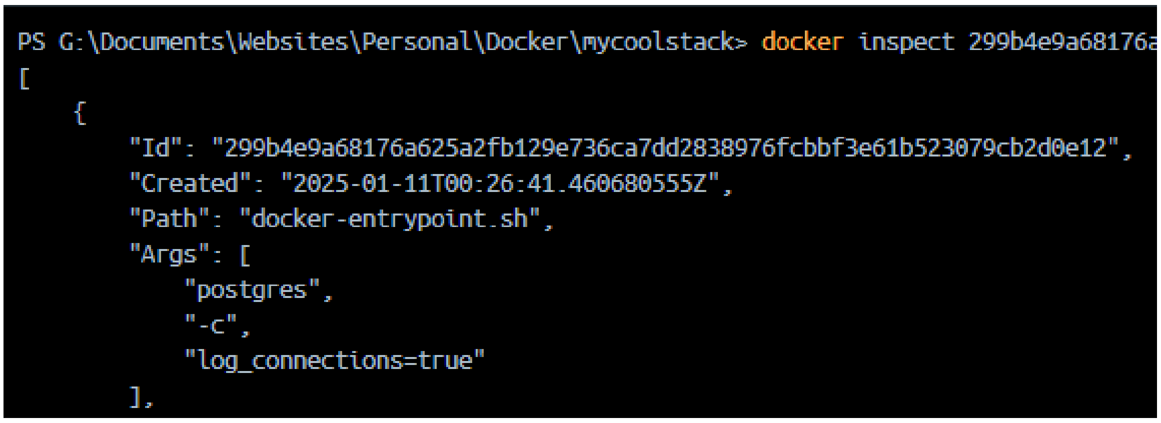
```
2025-01-11 00:26:22 172.18.0.1 - - [10/Jan/2025:22:26:22 +0000] "GET /
HTTP/1.1" 200 555 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:134.0)
Gecko/20100101 Firefox/134.0"
```

Now that we have a server handler, we can move on to the database. For this project I arbitrarily chose PostgreSQL, and it ended up being quite simple to set up. At this point the multiple service, multi-container application (or stack) is starting to take shape. The `compose.yaml` file is now going to list two services:

```yaml
# Compose file, docker 4.36.0
services:
  server:
    build:
      context: ./app/server
    restart: always
    volumes:
      - ./app/server/var/www/html/index.php:/var/www/html/index.php
    ports:
      - 2025:80 # Public HTTP Port
  db:
    image: 'postgres:latest'
    ports:
      - 5432:5432 # Default postgres Port
    restart: always
    command: ["postgres", "-c", "log_connections=true"]
    shm_size: 128mb # Set shared memory limit when using docker compose
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: example
      POSTGRES_DB: CoolStackDB
```
Listing 1.3.3: compose.yaml with a multi-container setup

This is one of the coolest things about Docker! Up until now we used a single endpoint for our application, but just like this we can keep adding more and more services that will talk to each other, where each one plays a crucial role in the stack but in the end they will all work together to make our idea possible.

This is also the first time we are using a base image straight in the compose file, instead of using a build context and a Dockerfile, however the process isn't much different. Imagine that there is a Dockerfile somewhere that has a bunch of directives that builds the container for us, and all we have to do is pull the image. Of course, we are allowed to make some configuration to these images, like the environment variables that set the credentials for the first user to be created for the database, as seen in Listing 1.3.3, with `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`. By using the attribute or element `command`, we are setting a directive for what the container is going to do after compose is finished and the container has started, and this behavior depends on whether an `entrypoint` has already been defined in the image's Dockerfile, as seen in Figure 1.3.1 below:



```
PS G:\Documents\Websites\Personal\Docker\mycoolstack> docker inspect 299b4e9a68176a
[
    {
        "Id": "299b4e9a68176a625a2fb129e736ca7dd2838976fcbbf3e61b523079cb2d0e12",
        "Created": "2025-01-11T00:26:41.460680555Z",
        "Path": "docker-entrypoint.sh",
        "Args": [
            "postgres",
            "-c",
            "log_connections=true"
        ],
```

Figure 1.3.1: db's entrypoint and command, in Docker terminal

When an `entrypoint` is defined, the `command` element in compose simply acts as arguments for `entrypoint`, otherwise it directly executes the process/binary on the first argument, with argument_1, argument_2, … argument_n as arguments. Here, `command: ["postgres", "-c", "log_connections=true"]` (from Listing 1.3.3) will pass as arguments to `docker-entrypoint.sh`. This is the file that initializes the container with settings provided from the environment variables like `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`. These particular arguments for `command` will be passed at the `exec "$@"` command in the `entrypoint` script , which essentially execute `postgres` with the `-c` configuration `log_connections=true`.

Generally with many services we want to enforce defaults when it comes to exposing ports, and PostgreSQL is one of those examples. While it is possible to choose a different port, often times it's not as simple as just exposing a new one in the compose file (at least not the container port), and require some digging to find where the configuration files are, and what the right variables are for port listeners, while nine times out of ten (not an accurate estimate, just a way of saying) the default port will be listed in the official image's Docker Hub page.

For our database container we could also include volumes to target the two configuration files `pg_hba.conf` and `postgresql.conf` in order to use our own configuration, however that would have to be done after the initialization, as this process requires that the data folder (the path the configuration files need to target) is empty, and fails otherwise. The reason we can compose, then add the volumes and compose again without losing the state of the container (the initialized data), is because this specific image creates an anonymous volume by default that targets the container path `/var/lib/postgresql/data/`, thus allowing data persistence, which is handy for this type of container but should not abolish the need for backups. An anonymous volume is one that does not specify a path on the host, only in the container, and its contents are managed and preserved by Docker.

A web interface to interact with and manage the database can be used, like adminer. This particular service will not provide any other role other than become the means to create a database structure/schema. The actual data that we will be writing into the tables will be done dynamically through PHP, or the server container, in communication with the database service. We can introduce this web interface service as shown in Listing 1.3.4:

```yaml
...
adminer:
  image: 'adminer:latest'
  restart: always
  ports:
    - 8080:8080
...
```

Listing 1.3.4: adminer service in compose.yaml

Last but not least is the solution for storing secrets, like database credentials. For a full overview of why this service is important and what the setup looks like, see chapter 2. For the time being, let's understand this step as completely optional, however incredibly beneficial as far as security is concerned. This service will be the space that the server will reach out to when calls to the database are required, providing the server with the credentials to actually communicate with the database.

```yaml
...
vault:
    image: 'hashicorp/vault:latest'
    restart: always
    ports:
      - '8200:8200' # Vault Web Port
      - '8201:8201' # Vault Cluster Port
    environment:
      - VAULT_DISABLE_USER_LOCKOUT=true # For development only
    volumes:
      - ./app/vault/config:/vault/config
      - ./app/vault/file:/vault/file
      - ./app/vault/logs:/vault/logs
    cap_add:
      - IPC_LOCK # Lock Memory
    entrypoint: vault server -config=/vault/config/vault.json
...
```

Listing 1.3.5: Secret storage service in compose.yaml

Briefly enough, referencing the above Listing (1.3.5), it can be mentioned that the Vault web port (8200) is the web interface we will use to set up the secrets, policies, and other settings, while internally the service will use the cluster port (8201) to facilitate jobs to the many workers or processes that handle client requests to Vault. Additionally, the three volumes respectively provide, a custom configuration to Vault, data persistence, and a path for logging if the feature is enabled later. For development we opt to lock the entire Vault process in memory which restricts it from swapping to disc (this is not recommended in production by HashiCorp where integrated storage is used instead; see HashiCorp, n.d.-c, for more detail), and disabling user lockout for multiple failed authorization attempts.

Currently, the service count in our compose stack is four, and only adminer is connected to the database because we are using postgres' default listening port, but otherwise the rest of the containers are logically isolated from each other. The next subchapter discusses how the connections are made using the internal network Docker provides by default, as well as client authentication for the database service.

Don't forget to `docker compose up -d`!

1.4 Container IP address Management

It’s no coincidence that we assign ports in `compose.yaml` in `host_port:container_port` format. The idea is that we can have multiple services per compose stack, and perhaps even multiple compose stacks per Docker environment. This assignment format helps distinguish uniquely accessible paths to each different service, both on the host side but also internally in the Docker network.

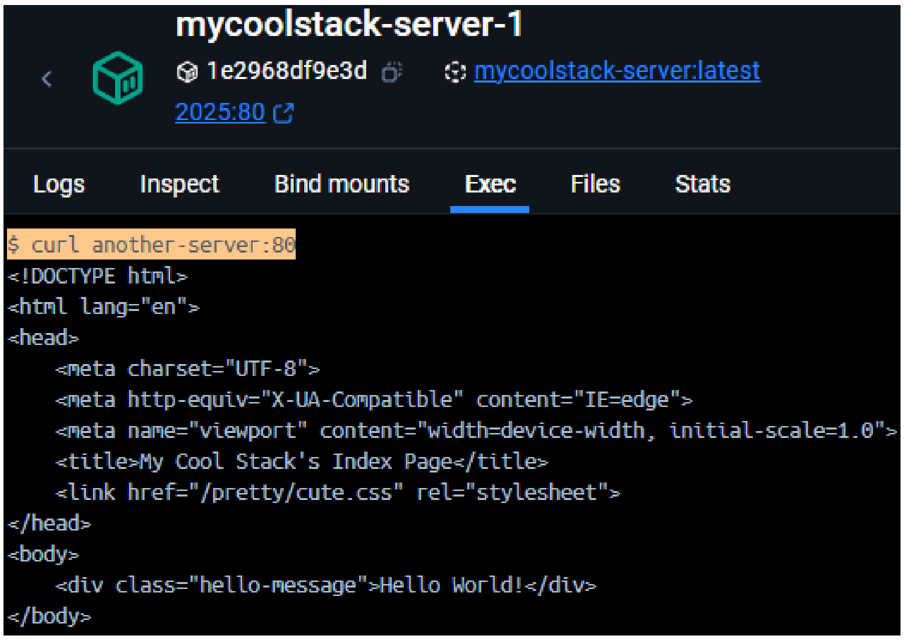
It’s clear that we need unique ports for services on the host side, because we can understand the concept of , for example, unique numbers identifying different houses in the same block – it makes delivering mail to the right address super easy. So why do we need to separate the host port from the container port? Let’s assume the following example:

```
services:
  server:
    build:
      context: ./app/server
    ports:
      - 2025:80 # Public HTTP Port
  another-server:
    build:
      context: ./app/server
    ports:
      - 3025:80 # Public HTTP Port
```

Listing 1.4.1: compose.yaml showcasing two similar services

The setup in Listing 1.4.1 is totally valid, although perhaps not incredibly useful unless two servers are truly necessary. Each one uses a unique host port (2025, 3025 respectively), but the same one in the container (80). Container ports and IP addresses is where internal Docker network magic (sort of) happens. Each service is normally reachable by other services from within the internal network, with the combination of an IP that’s assigned to them by Docker (these IPs are automatically set by default) and the container port that we set ourselves.

For a request that is received on host port 3025 for example, in order for Docker to forward this message properly, it’s going to use the IP address it set for `another-server`, and the port 80, so that could look like `172.18.0.4:80`. This address is what the internal network recognizes `another-server` as, and it can actually be invoked by using a more human readable format. Internally, any service in the compose stack identifies by its container name, followed by the port number.



```
$ curl another-server:80
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Cool Stack's Index Page</title>
    <link href="/pretty/cute.css" rel="stylesheet">
</head>
<body>
    <div class="hello-message">Hello World!</div>
</body>
```

Figure 1.4.1: Sending request from service-to-service

In the above Figure (1.4.1), we use the command `curl` to make a request from `server`, to `another-server`. The response from `another-server` might look familiar! It's the HTML we defined in subchapter 1.1, Listing 1.1.1, and it's exactly the response we'd get if we navigated to the IP address in a browser on the host that corresponds to `another-server:80`, which is `localhost:3025`.

With that information in mind, we'll now make temporary connection methods with server and each service it should be in direct communication with:

1. Database
2. Vault

Our server service (simply "server" from now on) will handle HTTP requests, and is expected to respond with data that it can fetch from the database. In order to connect to the database, it will first need to fetch the secrets (credentials to database) from the vault service (simply "vault" from now on).

Vault's internal address can be hardcoded as an environment variable in envvars (defined in subchapter 1.3, Listing 1.3.1) in server, and passed into each API call we'll make to vault. For example, one way to implement this is by setting the environment variable `VAULT_ADDRESS=vault:8200` then fetching it in `index.php` with `$vaultAddress = getenv('VAULT_ADDRESS');` then use this new variable to make the API call. Currently, vault has not been initialized, and is not accepting any API calls as it's in a state it calls "sealed". What that is and how vault is initialized in order to start accepting API calls from server will be discussed in chapter 2, but for now note that the variable we set above will become quite handy for server-to-vault requests, and is all we need to be able to address vault from server.

Environment variables are quite straight forward to set and use with PHP, however there is a small caveat. Every time we need to set a new environment variable in `envvars` we need to restart the service for it to be able to be used in PHP files.

Next, for the database, we can make use of a similar method to setup the address. First, let's take a look at what the connection function actually needs:

```php
$dsn = "pgsql:host=$hostname;dbname=$databasename;";
$pdo = new PDO($dsn, $username, $password, [
    PDO::ATTR_PERSISTENT => TRUE,
    PDO::ATTR_TIMEOUT => 5,
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
]);
```

Listing 1.4.2: Database PDO connection with PHP

From Listing 1.4.2 we can utilize a single environment variable that will map onto `$hostname` such as `PG_HOST_NAME=host.docker.internal`. This particular address can be used to automatically resolve to the host's IP address from within the Docker container. In such case, the host port for the database will be used. Alternatively, if we go with `PG_HOST_NAME=db`, then the internal network will be used, in which case the container port will be mapped to the connection. If no port is passed into the DSN string argument, the default is used, which is `port=5432`. The developer here has the choice to either define environment variables for each argument, like username, and password, then load them into local variables and use them to make the connection, or fetch them as secrets from vault. This article will be implementing the second of the two methods. More will be shared on setting up the database connection in chapter 3.

An important part of the database configuration as far as connections are concerned, is adjusting the client authentication model. The rules for this model can be changed in the

container path `/var/lib/postgresql/data/pg_hba.conf`. This file sets what authentication method will be required by the client based on what IP group they are in. There are useful comments in `pg_hba.conf` to help with what values are possible.
By default, in my configuration file there were three types of clients with `trust` method for authentication, one being local connections which the file defines as Unix domain socket connections, the second being `127.0.0.1/32` which maps to a single IP address, `127.0.0.1`, the classic loopback address (localhost), and the third one was the IPv6 loopback address `::1/128`. As an important warning, the `trust` method means that authentication can be done without a password for any user (if the user field is set to all), which is an obvious pitfall in security. It is advised that these are replaced with a secure authentication method like md5 or scram-sha-256 for any type of connection, for all databases and all users. The database connection will become an automated process for server calls after all, therefore if the reason to go with the `trust` method is for faster manual logins, the drawbacks on security are simply not worth it.

With each service having its address defined in the server container and ready to be used in the right functions, and client authentication rules set on connections to the database, we can conclude the first chapter and move onto working on the backend. The next chapter is dedicated to integrating the vault service to the server backend, but not without first explaining the initialization process, unsealing, setting up policies, creating secrets, and finally fetching them.

Remember, secret fetching isn't a necessary process, as we could've just hardcoded the credentials in the environment, but it's definitely the more secure way to do it. Keep an open mind to the idea, and you might be surprised by how handy it is.

2 Secrets with HashiCorp's Vault

When we are interested in elevating the security of our server, we have to think of what it is that we are really trying to protect. So far we have looked at building our containers as ephemeral as possible, meaning if our server was compromised, even if an attacker managed to bypass our security checks like having restricted permissions in the server space, the damage they could do to it would be easily replaceable. Generally, if the entrypoint is secure enough, we would say that anything that's connected to it is also secure, so for instance if the first access point the attacker can go through in our setup is the server, then securing that particular container as much as possible automatically hardens the security of every single service that's communicating with the server. Unfortunately, there is no checklist that will ever fully secure any device or software in the world, and this is why we address security as this granular and extensive process that almost every "moving" piece in the application can benefit from. So, back to the original question then: What is it that we are trying to protect?

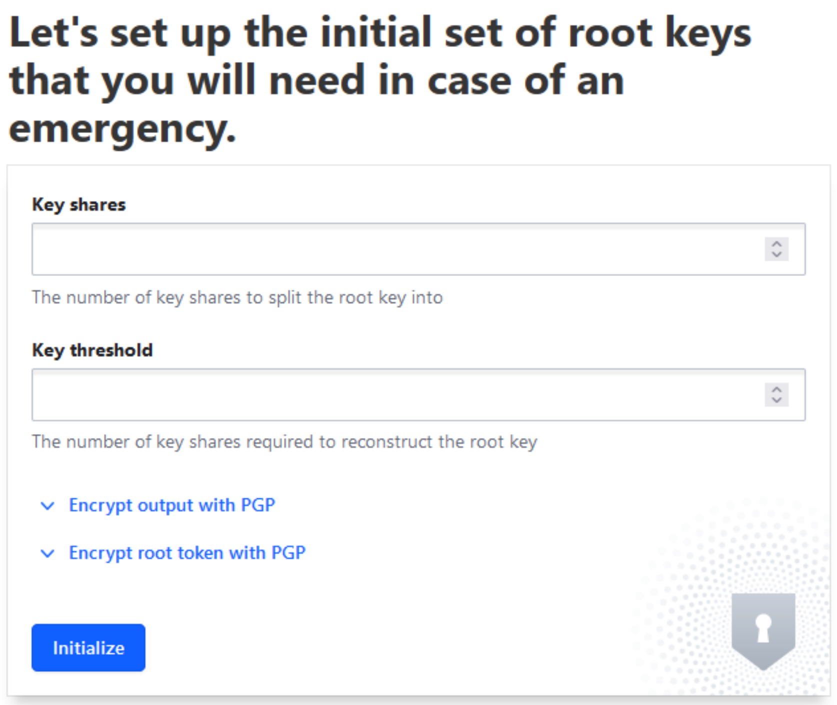
Let's assume that the attacker somehow has access to the database service, either through an interface we forgot to block access to, a port that was accidentally exposed on the modem/router level, or perhaps through the server container itself. Now, the compromised database is prone to be brute forced to have its username/password combo discovered, which is an interesting topic that unfortunately this Thesis does not cover (protecting against DDoS or brute force attacks), or the attacker could look for these credentials in the PHP functions that live in the server environment and if those are hardcoded (i.e. plaintext format) then very few things are in between the attacker and customer full names, addresses or emails, employee records, intellectual property patents or trademarks, and the list goes on and on.

This chapter focuses on making credential discovery a more demanding task for an attacker. The work that will go into hardening that aspect of security is by no means the ultimate solution to consider a system secure, and should be treated as one of the many steps an administrator can take to improve that security.

2.1 Vault Initialization

Having already containerized the vault service, there are a few steps we first need to take before we can start making API calls from the server container. Alongside the first initialization, a method to make vault as ephemeral as we can will be shown, utilizing Vault's snapshot feature.

In chapter 1, subchapter 1.3, Listing 1.3.5 we set the web interface port for the vault service to 8200 on the host, so that should allow us to navigate to `localhost:8200`. We should be met with a message telling us that "Vault is sealed", with the option to make a new Raft cluster. "Sealed" is a state Vault operates in after a cold boot (fresh start, or restart) or after "encountering an unrecoverable error" (HashiCorp, n.d.-b, Unsealing section), requiring that a number of keys are entered before it decrypts the content in its database. Technically, those keys are really only decrypting yet another key, called the root key, which in turn decrypts the actual data. Those keys are called key shares, and the administrator during initialization can choose the number of those shares that will exist, but also the minimum amount of shares required to unseal Vault:


Let's set up the initial set of root keys that you will need in case of an emergency.

Key shares

The number of key shares to split the root key into

Key threshold

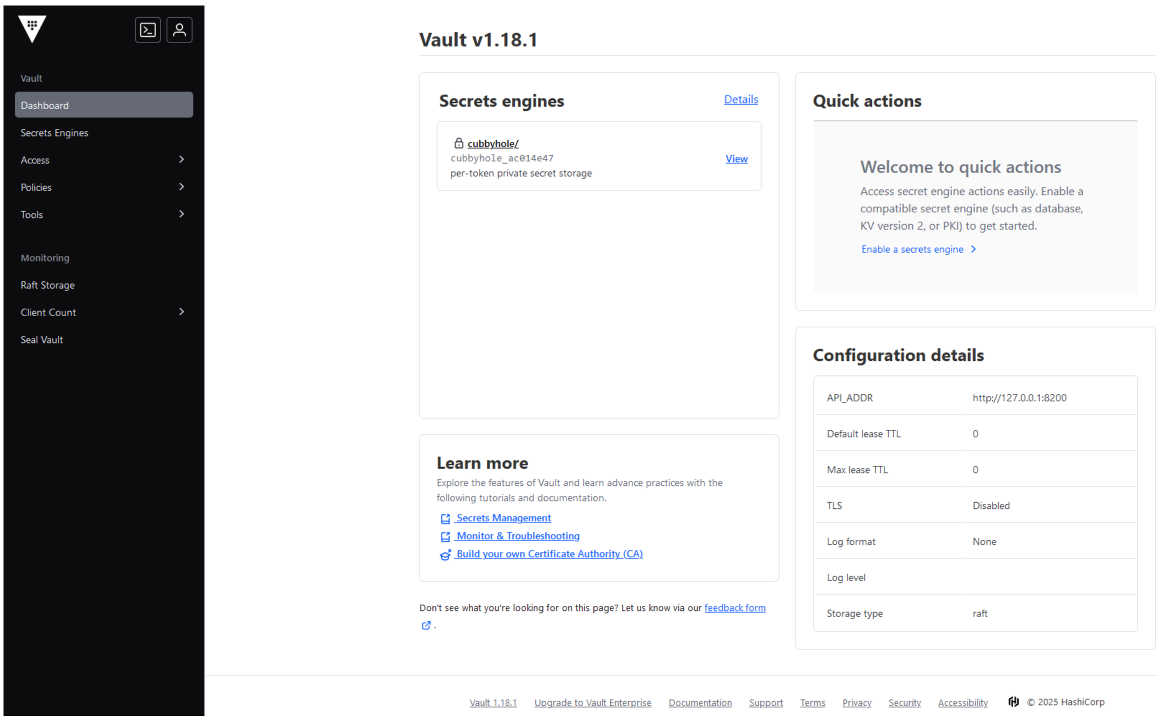
The number of key shares required to reconstruct the root key

Figure 2.1.1: Vault interface, initialization

As shown in Figure 2.1.1, we are also given the choice to encrypt the key shares and root key outputs using a PGP public key. To decrypt the output we'll need our private PGP key. In the case we don't want the output for the values of the key shares and root key to be encrypted, we can simply initialize. In either case, Vault generates the key shares and root key for us, and they become retrievable (or downloadable as a file) on the next screen.

How each administrator chooses to store these keys is completely up to them, but note that if a significant amount (number exceeding the key threshold) of unseal keys are lost then Vault will not be able to be decrypted, with all its data essentially becoming unrecoverable.

To get into the main interface, we first need to unseal Vault, then provide the root key to log in.



Vault v1.18.1

Secrets engines

Details

cubbyhole/
cubbyhole_ac014e47
per-token private secret storage

View

Quick actions

Welcome to quick actions

Access secret engine actions easily. Enable a compatible secret engine (such as database, KV version 2, or PKI) to get started.

Enable a secrets engine

Configuration details

API_ADDR	http://127.0.0.1:8200
Default lease TTL	0
Max lease TTL	0
TLS	Disabled
Log format	None
Log level	
Storage type	raft

Learn more

Explore the features of Vault and learn advance practices with the following tutorials and documentation.

Secrets Management
Monitor & Troubleshooting
Build your own Certificate Authority (CA)

Don't see what you're looking for on this page? Let us know via our feedback form .

Vault 1.18.1 Upgrade to Vault Enterprise Documentation Support Terms Privacy Security Accessibility © 2025 HashiCorp

Figure 2.1.2: Vault interface, logged in

The landing page after providing the root key looks like what's shown in Figure 2.1.2. All of the API has now been unlocked. An exhaustive list of that API can be found under Tools -> API Explorer, which can be invoked either right there on the interface (e.g. for testing), the vault container through CLI, or PHP with HTTP requests, which is the case we'll be exploring in this chapter.

In later subchapters, we will be creating policies, an authentication method, a secret engine, and the secrets themselves. All of these settings can be bundled into a package and re-used in other projects/compose stacks, using Vault's snapshot feature. When we have fully prepared Vault for a snapshot, we can download that bundle of settings by going to Raft Storage -> Snapshots -> Download. The caveat is that in order to restore it on a freshly initialized Vault container, we'll need the original key shares in order to unseal Vault, and the original root key to be able to log in, which makes sense from a security perspective, but obviously retracts from the ephemeral nature of Docker containers. Vault in my experience was the least ephemeral container (or maybe on par with the database one), but even then, the most one has to do to get it fully functional is:

1) initialize,
2) unseal,
3) log in,
4) restore with snapshot, and
5) unseal with original keys

Perhaps there is a smart way to script all of these actions in Vault's entrypoint file, and fully automate that process too, but otherwise the snapshot feature reduces the amount of setup required by a massive margin.

Note that the only thing not saved in the snapshot are tokens that were created with a lease time (i.e. expiring tokens), so if these are used then add a few more steps above to create them again manually. What these are and how we use them in our setup will be shown in later subchapters.

2.2 Vault Policies

The main idea with Vault API, is for endpoints to be accessible only by processes that have permissions to do so, and ideally only one type of process will have permission to access one type of endpoint. For this purpose, we can say that these processes will be the PHP functions that call an API, endpoints will be each unique API route, and permissions will be the policies we set that allow for a process to have access to an endpoint.

The way we'll be able to link a process in the server container (it's where PHP functions live) with a policy in vault, will be done using tokens. Let's look at an example process in index.php, in server:

```php
<?php
require_once __DIR__ . '/../vendor/autoload.php';
use GuzzleHttp\Client;
use GuzzleHttp\Exception\RequestException;

$vaultAddress = getenv('VAULT_ADDRESS');
$vaultToken = getenv('VAULT_TOKEN_SCRT');
function getSecret(string $vA, string $vT) {
    $client = new Client([
        'base_uri' => $vA,
        'headers' => ['X-Vault-Token' => $vT],
    ]);
    try {
        $response = $client->get('/v1/kv/data/databasesecret');
        $body = json_decode($response->getBody(), true);
        return $body['data']['data'];
    } catch (Exception $e) {
        die("Failure: <h2><b>Message:</b></h2> " . $e->getMessage() .
PHP_EOL);
    }
}
$secret = getSecret($vaultAddress, $vaultToken);
```

Listing 2.2.1: Invoking vault API from server

Using Guzzle, we can invoke vault API from server, at the address we defined in chapter 1, subchapter 1.4 for `$vaultAddress`, which was simply `vault:8200`. Referencing the API on Vault's web interface, we get some important details on how we need to structure the request to get the right data. Those are as follows (as seen in Listing 2.2.1):

1) GET request - `$client->get()`
2) API route - `$vaultAddress + '/v1/kv/data/mysecret'`
3) Token - `$vaultToken`
4) Structure of response (JSON) - `$body`

Remember how I said that we would be using a token as a means to link the process (the function in Listing 2.2.1) with a policy in vault? This token, `$vaultToken`, is how that's done. A simple way to understand the function + token pair, is to think of that token as the root key that Vault generated for us during initialization. That key has all of the permissions it needs to access any API endpoint at vault. Providing the root key to this particular API endpoint grants us access to the data the endpoint is holding, for this example a secret (`/v1/kv/data/mysecret`). If the key, or token, didn't have the right permissions, vault would respond with "403 Permission denied", which is a good thing because we don't want a potential attacker to be able to access our secrets. The only problem with using the root key as the token however, is that we are betraying the model we set initially in this subchapter: one type of process will have permission to access one type of endpoint.

To anyone that wondered about the code from Listing 2.2.1, specifically how we hardcode the token into the environment, good job! That is a big issue, and we'll explore a different way to read tokens soon.

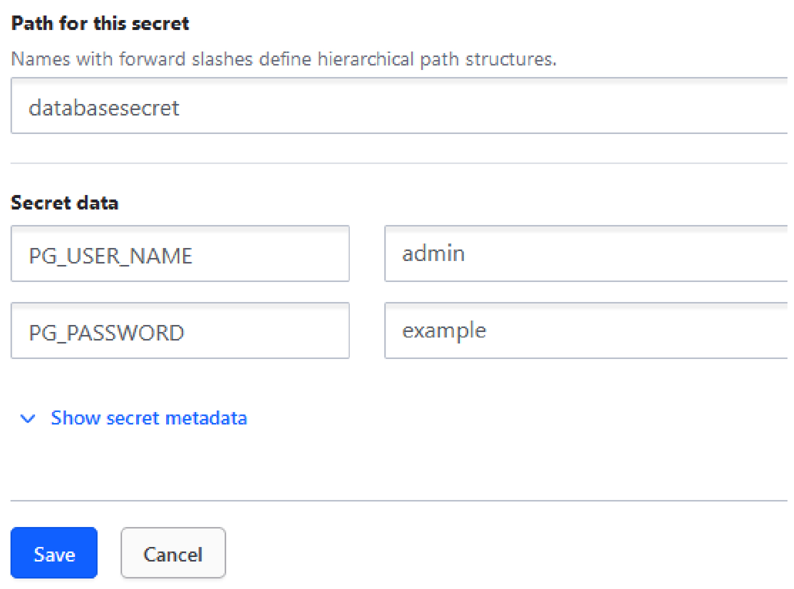
Let's look at how we can create a token for our process. To do this, we will need a combination of actions on Vault's interface, and commands through the CLI. To create a token we'll first need to create a policy to attach it to, and to create a policy we need to have an existing secret, and in order to create a secret we need to create a secrets engine. That's a mouthful, so let's write that in a nicer format:

Secrets engine `->` Secret `->` Policy `->` Token

Starting with the secrets engine, we will choose one of the generic options Vault offers, called kv. This engine follows the traditional `key:value` format, which perfectly covers the use case we need it for. It also features versioning, where if a value for a particular key is updated, the old version is not overwritten by the new one but instead stored to be accessed by its version number. This feature will not be getting any use in this article. To create the engine, we can do so through the interface, by going to Secrets engines `->` Enable new engine `->` KV `->` Enable engine.

On to the secrets themselves. We should understand each secret as a "folder" containing multiple `key:value` pairs, because after all, fetching a secret from Vault returns a JSON string, containing all of those pairs with one API call. If we desire to store many different types of secrets in Vault, then it makes sense to separate these so that when we create policies, we can target one secret path at a time. While still on the interface, we can create a secret at Secrets engines `->` kv (or whatever we named our engine) `->` Create secret. Here, we can set the name of the secret group (path), and a couple of `key:value` pairs.

We are trying to secure the database, so these pairs should cover at least the database username & password (shown in Figure 2.2.1), and maybe even the database name:


Path for this secret

Names with forward slashes define hierarchical path structures.

databasesecret

Secret data

PG_USER_NAME	admin
PG_PASSWORD	example

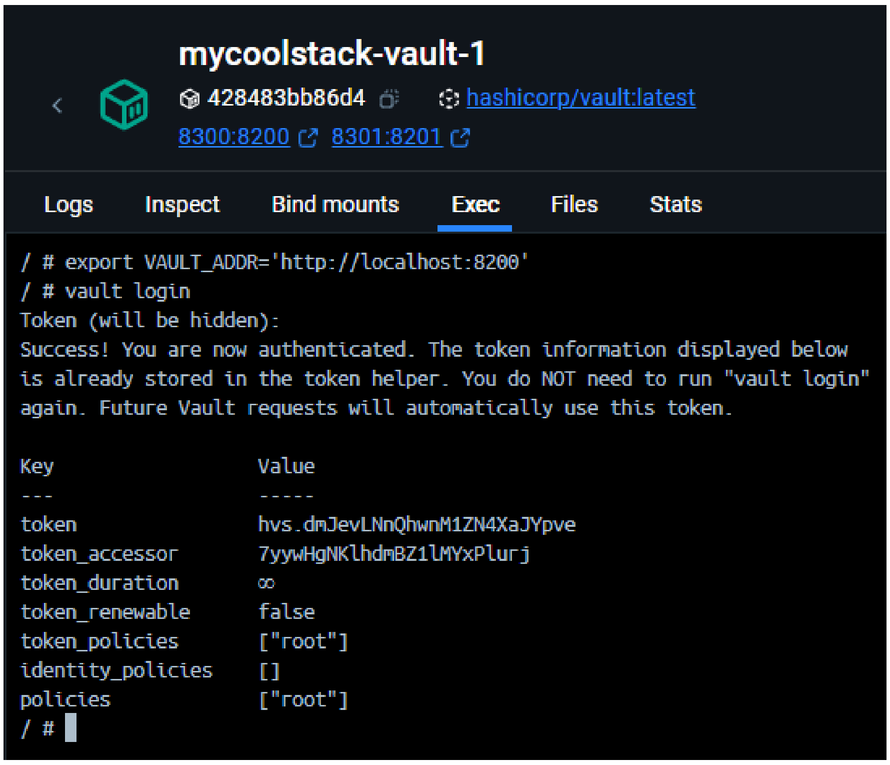
Figure 2.2.1: Example Vault secret, in Vault's interface

Now that we have the secret ready, we have a path we can target a policy at, and after that only a token is left to apply that policy to. For the policy, on the interface, let's go to Policies -> Create ACL Policy. These can be written in HCL (HashiCorp Configuration Language), so a proprietary format, but it's got a pretty human readable nature. Much like setting permissions in our Linux-based containers, what we intend to do is set the least privilege possible that still allows us to do the operation we need, but nothing more. To fetch a secret, all we need is the "read" capability on the path of the secret, as shown in Listing 2.2.2:

```
path "kv/data/databasesecret" {
  capabilities = [ "read" ]
}
```

Listing 2.2.2: Policy example in Vault

Simple! Creating the policy with this rule, gives us the proper permission that the process needs to fetch the secret and nothing more. We can now issue a token with this policy attached. This will be done in the CLI, in the container console. All we need to create a basic token, is the name we gave to the policy. First, however, we need to authenticate with Vault. In the console, we do this by defining `VAULT_ADDR`, with the following command: `export VAULT_ADDR='http://localhost:8200'`. Note here, that this address refers to the container's address, as well as the port, NOT the host. Then, we can log in using our root key, with `vault login <root_key>`. For extra security, we can execute `vault login` without providing the root key at first, and type it when it's asked in a new line. This ensures that the console doesn't capture the key in command history (which is plaintext):



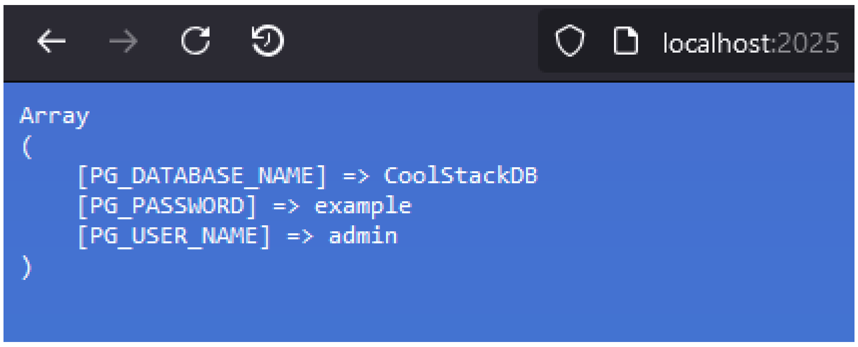
```
/ # export VAULT_ADDR='http://localhost:8200'
/ # vault login
Token (will be hidden):
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                  Value
---                  -----
token                hvs.dmJevLNnQhwnM1ZN4XaJYpve
token_accessor       7yywHgNKlhdmBZ1lMYxPlurj
token_duration       ∞
token_renewable      false
token_policies       ["root"]
identity_policies    []
policies             ["root"]
/ #
```

Figure 2.2.2: CLI login process, in vault

The login process as described, is shown in Figure 2.2.2. We can now generate a token for the policy we created earlier, with `vault token create -policy=database-policy -no-default-policy`. Ensuring that no more permissions than necessary are granted, we remove the default policy group from being automatically attached to the generated token. Without providing any additional flags, the token will be renewable, and have an expiry time of a month (768hrs) which is the max token duration by default. While the maximum duration can be configured to be higher than that, it is discouraged, and instead short-lived tokens should be generated in our application and automated to be inserted in the functions that need them. This article does not cover automating token generation.

These are the tokens that were briefly mentioned in the last subchapter (2.1), being the only data that does not get saved in a snapshot.

With a token now generated, it can be used as the means to access the secret. We can export it in the environment variable `VAULT_TOKEN_SCRT`, from listing 2.2.1 and run index.php, in server. The credentials for the database are stored in `$secret` as an array, where each value can be read using the corresponding key in the secret (Figure 2.2.3).



```
Array
(
    [PG_DATABASE_NAME] => CoolStackDB
    [PG_PASSWORD] => example
    [PG_USER_NAME] => admin
)
```

Figure 2.2.3: Server response with secret fetching

As is evident by the populated values in the array, our token is valid, and it has enough permissions to read the secret that the secrets engine is managing. We have successfully linked a process (in this example fetching a secret), with a token. But the problem is that the token currently is in plaintext format in the environment, and you've probably already guessed it, that essentially defeats the purpose of using Vault entirely.

This problem already peeks into one of the biggest issues with secret management which is called the Secret Zero problem, and it's what will be discussed next.

2.3 Secret Zero problem

If you want to hide something, where do you hide the thing that you hid the first thing in?

This is what Secret Zero (SZ from now on) basically is: the first secret, or sensitive piece of information, that's required in a chain of security safeguards. Right now, the SZ for our Vault setup, is a token that has permission to access a secret. The token being the SZ is not an issue in and of itself, but rather how it's retained in the server so it can be used by the appropriate function (process). What we need is a safe way to store it, as right now it’s in plaintext format, as an environment variable in envvars.

Instead of storing SZ in server, we can completely offload it to a different service, and safely deliver it to server when it needs to fetch the secret. This, is what we call a Trusted Entity – a service or system that we have trusted to manage sensitive information on our behalf.

What we are particularly interested in, is the secure delivery aspect of this service, and for this we will be using Vault’s Response Wrapping mechanism. When server is ready to accept the token, it can signal the Trusted Entity to begin its process, which starts with wrapping the token, SZ, then placing it in a “sink”, which is a temporary location that server and Trusted Entity share (this could be a shared volume, for example), and then finally server can unwrap the token and use it to fetch the secret.

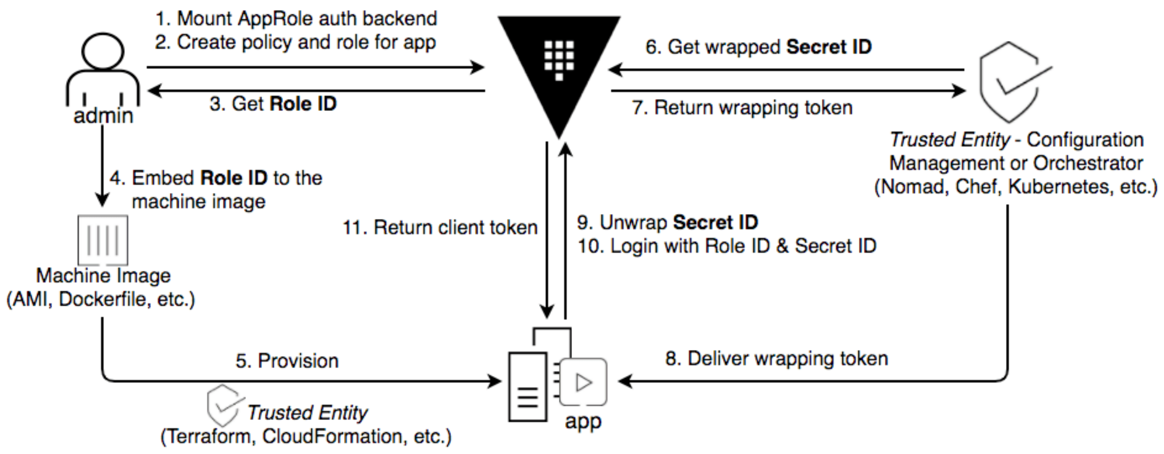
The inner workings of Response Wrapping are interesting, especially when we understand what a wrapped token is. When we ask Vault to wrap SZ for us, it is first encrypted into Vault’s private secrets engine, called “Cubbyhole”, then for a single-use, short Time-To-Live (TTL) token to be issued that has authorized access to use one of Vault’s encryption keys to decrypt the data, which is SZ itself (see HashiCorp, n.d.-a, for more detail).

The single-use nature of these tokens is particularly useful, because it ensures that exactly one recipient gets to see the contents of the wrapped token, continuing operation for a normal use scenario, but providing the option to notify the system if a token is attempted to be unwrapped more than once.

These rules help understand how our system ought to work in theory, and set precedent for designing yet another service that’s supposed to play a very important role. In practice, we need a container that can communicate with vault, much like server can right now. For this purpose, we are going to need a way to send HTTP requests, as well as to define the “sink” we discussed earlier as the shared volume between this Trusted Entity and server to store the wrapped token vault responds with. The HTTP request can be done in a script that’s triggered whenever server signals Trusted Entity, and runs a function, a new process, that wraps SZ, and places it into the shared volume. We can utilize the curl command in Trusted Entity to make these HTTP requests in the script.

Before we set up the new service, we will be switching the SZ, from a static token which it currently is, to a dynamically generated ID that has a single use just like the wrapping token, and a much shorter TTL. These IDs can easily be generated with Vault when using an authentication method called AppRole.

We won't be completely abandoning tokens, as these still have their use in our setup, but their role will be slightly different.



1. Mount AppRole auth backend
2. Create policy and role for app
3. Get **Role ID**
admin
4. Embed **Role ID** to the machine image
Machine Image (AMI, Dockerfile, etc.)
5. Provision
Trusted Entity (Terraform, CloudFormation, etc.)
6. Get wrapped **Secret ID**
7. Return wrapping token
Trusted Entity - Configuration Management or Orchestrator (Nomad, Chef, Kubernetes, etc.)
8. Deliver wrapping token
9. Unwrap **Secret ID**
10. Login with Role ID & Secret ID
11. Return client token
app

Figure 2.3.1: AppRole authentication workflow in HashiCorp's Vault

How AppRole is going to be replacing the existing token, is by splitting up into two parts, called Role, and Secret IDs (imagine these as username & password) like pictured in Figure 2.3.1. This model makes it possible to use a static part like the token was, in combination with a dynamic component that's more flexible because it can be single use, and have a very short TTL. The Secret ID being this dynamic component, offers unique IDs that can be configured to be short lived, compared to the token that only changes when the expiration occurs.

In vault's console, while logged in, we can enter the following commands (Listing 2.3.1) to get started with AppRole:

```
vault auth enable approle
vault write auth/approle/role/my-role token_type=batch secret_id_ttl=2s
token_ttl=2s token_max_ttl=5s secret_id_num_uses=1
```
Listing 2.3.1: Enable AppRole through CLI, in vault

If we are already authenticated with vault in the container's CLI, we should see a success message for both of those commands. With this command, we are creating an AppRole called `my-role`, and we are restricting this AppRole's Secret IDs to have a 2 second TTL, and allow AppRole to create tokens with a TTL of 2 seconds as well. These tokens inherit the policies that AppRole has, so we can attach the policy we already made for the token we are now replacing, to the AppRole method:

```
vault write auth/approle/role/my-role token_policies="database-policy"
token_ttl=2s token_max_ttl=5s
```

Listing 2.3.2: Attaching a policy to AppRole

Now we have AppRole enabled, with the original policy attached to it that allows the tokens AppRole creates to read databasesecret, which contains the database credentials. To summarize, we replaced the single, static, long-lived token we created in the last subchapter, with AppRole, which creates tokens after combining the Role ID and Secret ID, that together fetch the secret. This choice was made so that we can provide server short-lived IDs that are single use, compared to our long-lived, infinite uses token. In essence, it's to bolster security. In the setup that follows, the Role ID is treated as a constant, and is hardcoded in the environment in server, like token was before being offloaded to the Trusted Entity, and the Secret ID will be what really replaces the token in the Trusted Entity. For AppRole's true potential to be adapted, the administrator can instead of hardcoding the Role ID in server, provision it with Terraform, or Ansible. This usage of AppRole is outside of the scope for this subchapter.

Obtaining AppRole's Role ID can be done in the CLI using the command in Listing 2.3.3:

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

Listing 2.3.3: Obtaining the Role ID

The value vault returns will be hardcoded into the environment in server. For the Secret IDs we will create a script in Trusted Entity that asks vault to return a wrapped Secret ID:

```
path=$1
vAddress="http://vault:8200"
vToken="hvs. ... "

RESPONSE=$(curl -s \
  --header "X-Vault-Token: $vToken" \
  --header "X-Vault-Wrap-TTL: 2s" \
  --request POST \
  $vAddress/v1/auth/approle/role/my-role/secret-id)

WRAPPED_SECRET_ID=$(echo $RESPONSE | jq -r .wrap_info.token)
echo "$WRAPPED_SECRET_ID" > "/pipe/wsID_$path.txt"
```

Listing 2.3.4: Wrapped Secret ID fetching script

In listing 2.3.4 we are making an API call to vault at the address `vAddress`, and a token `vToken`. It was mentioned earlier that tokens wouldn't be completely abandoned! Here, instead of hardcoding the root token, we can introduce a token with just the permission to wrap tokens, so even if an attacker somehow managed to escape the server container and gained access to Trusted Entity, they wouldn't get access to a token with global permissions on vault. We will create this token in the same way we created the original one, but instead of attaching the policy to read a secret, we'll create a new policy that can update Secret IDs for the specific AppRole `my-role`. The update capability only allows the token to create new Secret IDs, including wrapping them, but not reading the actual Secret ID.

```
path "auth/approle/role/my-role/secret-id" {
  capabilities = ["update"]
}
```

Listing 2.3.5: Wrapping policy, in Vault interface

With the policy created as shown in Listing 2.3.5, we can now attach it to a new token:

```
vault token create -policy=token-wrapper-policy -no-default-policy
```

Listing 2.3.6: Creating a token with wrapping policy through CLI, in vault

We can copy the token generated from the command in Listing 2.3.6 into the script for `vToken`. In Listing 2.3.4 shown in the previous page, we are writing the response from vault (the wrapped Secret ID) into a `txt` file, in a folder called `pipe`. This is our "sink", the shared volume between Trusted Entity and server. And speaking of Trusted Entity, it's about time we created this service:

```
...
trustedentity:
    build:
      context: ./app/trustedentity
    env_file:
      - ./app/trustedentity/etc/apache2/envvars
    restart: always
    volumes:
        - tmpfs-shared:/pipe
    depends_on:
      - vault
...
```

Listing 2.3.7: Trusted Entity service, in compose.yaml

In Listing 2.3.7, along with creating the Trusted Entity (from now on trustedentity), we are assigning the shared volume in the container. This one will also be assigned to server, but it needs to be defined at the bottom of `compose.yaml`, like in Listing 2.3.8:

```
...
# after services
volumes:
  tmpfs-shared:
    driver: local
    driver_opts:
      type: tmpfs
      device: tmpfs
```

Listing 2.3.8: Defining the shared volume, in compose.yaml

These types of volumes are used in cases where we are not interested in retaining the data after a restart. These tmpfs mounts are not binding host filesystems to the container, but instead are created in memory and are wiped when the container shuts down or restarts.

To be able to use the script in Listing 2.3.4 from two pages ago, we need the following binaries:

1. Curl - To invoke vault API
2. Jq - To format the response

Additionally, we are going to need a way for server to be able to reach trustedentity and signal it to generate a wrapped Secret ID. The way I implement this bridge, is with a messaging broker called RabbitMQ, which will be utilized with PHP to make Remote Procedure Calls (RPC) from server to trustedentity. For this, we will need another Apache server, since the container will be listening for messages. More details on the RPC mechanism in the next subchapter. For now, we will treat the signaling mechanism as a black box (an unknown/undefined system), that simply invokes or triggers the wrapped Secret ID script.

In the next Figure, both the Apache server and RabbitMQ get installed in trustedentity for consistency:

```
# syntax=docker/dockerfile:1
# TrustedEntity

FROM php:8.2.10-apache
# JQ, Curl, Zip
RUN apt-get update && apt-get install -y jq \
    curl \
    libpq-dev \
    libzip-dev \
    zip \
  && docker-php-ext-install zip \
    sockets \
    bcmath
# Install Composer
RUN curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer
ENV PATH="$PATH:/usr/local/bin"
# RabbitMQ
RUN composer require php-amqplib/php-amqplib
# Build + Ownership
WORKDIR /
COPY --chown=www-data:www-data . .
# Perms
RUN chmod 0400 /var/www/html/mq-consume.php
RUN chmod 0500         /vault/vlt-secure.sh
# Production Configuration
RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"
# Switch User
USER www-data
```

Listing 2.3.9: Trustedentity's Dockerfile

As shown in Listing 2.3.9, in order to use RabbitMQ we need the AMQP library for PHP, the binaries `sockets` and `bcmath`, and Composer. To install Composer we need the zip binary. Our wrapper script lives in `/vault/vlt-secure.sh` and will be invoked by `/var/www/html/mq-consume.php`, which is the container's message listener.

When the script is invoked, the JSON returned by vault looks like Listing 2.3.10:

```
  ...
  "wrap_info": {
    "token": "hvs.CAESIM0DFK0 ...",
    "creation_path": "auth/approle/role/my-role/secret-id"
  }
  ...
```

Listing 2.3.10: Vault's response to the script

The wrapped token value is captured with `WRAPPED_SECRET_ID=$(echo $RESPONSE | jq -r .wrap_info.token)` (from Listing 2.3.4) and placed in a `txt` file in `/pipe/` that is unique to the client that requested it, based on their session in server.

Now that we have extracted the wrapped Secret ID from trustedentity, we can read it from the `txt` file in the server container, and proceed to unwrap it. This process does not require a token to authenticate with vault, as every wrapped token has the permission to call the unwrap API on itself:

```
function unwrap(string $vA) {
    $path = '/pipe/wsID_'. $_SESSION['session_c_id'] .'.txt';
    $wsID = trim(file_get_contents($path));
    $client = new Client([
        'base_uri' => $vA,
        'headers' => ['X-Vault-Token' => $wsID],
    ]);
    try {
        $response = $client->post('/v1/sys/wrapping/unwrap');
        $body = json_decode($response->getBody(), true);
        return $body['data']['secret_id'];
    } catch (Exception $e) {
        die("Failure: <h2><b>Message:</b></h2> " . $e->getMessage() .
PHP_EOL);
    }
}
$rsID = unwrap('vault:8200');
```

Listing 2.3.11: Unwrap API call, in server, vault-key-unwrap.php

In listing 2.3.11, the unwrap function takes into consideration a custom ID that is set on every session in the server, and opens the file that corresponds to the right client. Using Guzzle to make an HTTP request to vault, we put the wrapped token we just captured in the request's header as the `X-Vault-Token`, and use the POST method to call the right API endpoint `/v1/sys/wrapping/unwrap`. In `$rsID` we should have the unwrapped Secret ID.

The last step to retrieve the actual secret, is to authenticate to vault using the AppRole. As a reminder, we hardcoded the Role ID in the environment in server, and we just obtained the Secret ID after unwrapping it successfully. The next API call combines those two to return a single token that will be used to authenticate to vault again, and fetch the secret:

```
...
$response = $client->post('/v1/auth/approle/login', [
    'json' => [
        'role_id' => $rID,
        'secret_id' => $rsID
        ]
]);
$body = json_decode($response->getBody(), true);
return $body['auth']['client_token'];
...
```

Listing 2.3.12: Authenticating to vault with AppRole

The only unique thing in the API call in Listing 2.3.12 compared to the rest so far, is how we are sending the data in the request. This is the JSON data option, which sets the `content-type` header of the HTTP request to `application/json` (Guzzlephp, n.d.-a), so vault can recognize the format.

Again, from the AppRole authentication, we are supposed to retrieve a token that has inherited the policy from AppRole, allowing it to access the secret.

For having a catchy name, “Secret Zero” turns out is a lot more about the technical side of managing secrets, rather than some fantastical story about perhaps replacing secrets with AI that “remembers” them for us. Who knows, maybe that will be done in the future. Nevertheless, while the attempt to solve the problem wasn’t the most robust, and definitely not having fully utilized all of the tools available from Vault, I am happy with the current setup but surely looking forward to improving on this in the future, particularly by provisioning the Role ID with Terraform, and revisiting the plaintext format idea for the shared volume.

The only thing left to discuss in this chapter, is the mechanism that triggers the wrapping script.

2.4 RabbitMQ RPC Approach

The messaging broker that’ll be utilized in this subchapter called RabbitMQ, will involve communication between multiple containers. The goal to be accomplished, is rather simple: Invoke a service to engage in a procedure, remotely from another service. Of course, I am referring to what we saw previously, in subchapter 2.3, with a very particular script that needs to be executed when a client requests the Secret Zero, which is the Secret ID for the AppRole authentication. In the previous subchapter, we also installed the AMQP library for PHP in trustedentity in order to use RabbitMQ function calls. At this point, it is assumed it is similarly installed in the server container, and that we have containerized RabbitMQ as a service in our compose stack.

The main reason I chose a message broker for the invocation, is because I originally ran into an issue with HTTP requests from server to trustedentity, particularly related to timing. Specifically, I was sending a basic POST HTTP request from server to execute the script on trustedentity, but the server container would keep moving on to the next block of code before trustedentity could finish with the script, resulting in the wrapped token not being properly obtained.

For those that are unfamiliar with the issue, it stems from the differences in asynchronous and synchronous programming. With the setup I was going for, I was creating an asynchronous task that would run in parallel with the process that called it, which had an undesirable effect in the expected flow. To fix this, the process calling the remote task, or procedure, would have to wait until the job is done remotely so that it can continue with the next block of code locally. In other words, I needed a way to run that remote procedure synchronously in respect to the process that called it.

Jumping ahead of myself here with this, but this is the exact point that forces the synchronous flow, essentially fixing the issue described:

```php
// -- Publish to remote --
$channel->basic_publish($msg, 'amq.direct', 'wake_iv');
// -- End publish to remote --
$response = null;
// -- Consume from remote --
$channel->basic_consume( ... ,
    function (AMQPMessage $msg) use (&$response, $correlationId) {
        if ($msg->get('correlation_id') === $correlationId) {
            $response = $msg->body;
        }
    }
);
while (!$response) $channel->wait();
// -- End consume from remote --
```

Listing 2.4.1: RabbitMQ channel with PHP, in server, mq-produce.php

The code block shown in Listing 2.4.1 is responsible for both invoking the response wrapping script in trustedentity, and forcing the synchronous flow we need. For the latter, a single line is causing a suspension: `while (!$response) $channel->wait();` where `$response` is at first initialized as `null`, and in the remote procedure space (consumer from now on) we just make sure to populate the response's body to break the loop in the message sender (producer from now on) and continuing the flow.

To start the conversation between server and trustedentity, a message is published to a channel with `$channel->basic_publish($msg, 'amq.direct', 'wake_iv');` where the message itself, `$msg`, contains a custom ID generated with the client's session, a specific exchange defined on RabbitMQ as `'amq.direct'`, and a routing key that helps find the right message queue called `'wake_iv'`. Technically, the producer is also expecting a message with `$channel->basic_consume()`, making it a consumer too, but the distinction between the two, at least theoretically, is based on the context of which container is responsible for starting the conversation. To be clear then, let's set the precedent from here on out as the server being the producer, and trustedentity being the consumer.

Understanding what the channel, exchange, routing key, and queues are, might be easier if we can think of the message delivery with an example in real life: At a post office, all of the mail is gathered for the day and taken to a distribution center. Here, it's decided how and where the message is supposed to be delivered, as mail gets sorted based on information about the destination, like the ZIP code (see Oneupme, 2024, for more detail). In RabbitMQ, the distribution center is the exchange, the ZIP code is the routing key, the entire route the mail will take is the channel, and the queue is the recipient's mailbox. The `'amq.direct'` exchange in particular, decides that messages with a routing key go to a specific queue. This mapping is called binding. So when a message is produced and goes out in the channel, it gets directed to a specific queue based on the exchange referencing routing key bindings. On the receiving end, we can configure a consumer to be listening for messages on that specific queue.

It should be clear enough where the messages are going, and the consumer should know exactly where to be listening for them, but there is one caveat. How will the consumer know where to send the returning message to confirm with the producer that the procedure was completed? This time, we can't just dump the message in a queue, because we need a specific client to receive the confirmation.

```
$callbackQueue = $channel->queue_declare(
        '',    // Unique server-named queue
        false, // Non-durable
        false, // Not exclusive
        true,  // Auto-delete
        false) // Passive declaration
        [0];   // Retrieve queue name
$correlationId = uniqid();
$msgOutgoing = [
        'session_c_id'  => $_SESSION['session_c_id'],
    ];
$msgOutgoing = json_encode($msgOutgoing, true);
$msg = new AMQPMessage($msgOutgoing, ['correlation_id' => $correlationId,
'reply_to' => $callbackQueue]);
```

Listing 2.4.2: Signing a message, in server, mq-produce.php

In Listing 2.4.2 we are using two identifiers in the message for the consumer, trustedentity, to know where to return it. First, a server named queue is declared, and as such will have a unique name. This is done by leaving the name in the first argument of `queue_declare()` as the empty string, and can be obtained by accessing the first element of the array returned by the function with `[0]`. Second, an ID is generated to be able to sign the message with, named `$correlationId`. This is the ID that the while loop from Listing 2.4.1 from two pages ago is expecting the message to contain, as well as the body of that message to not be `null` which we take care of on the consumer side.

With that said, and before the consumer setup is shown, let's zoom out a little and take a look at how things are on the macro level:

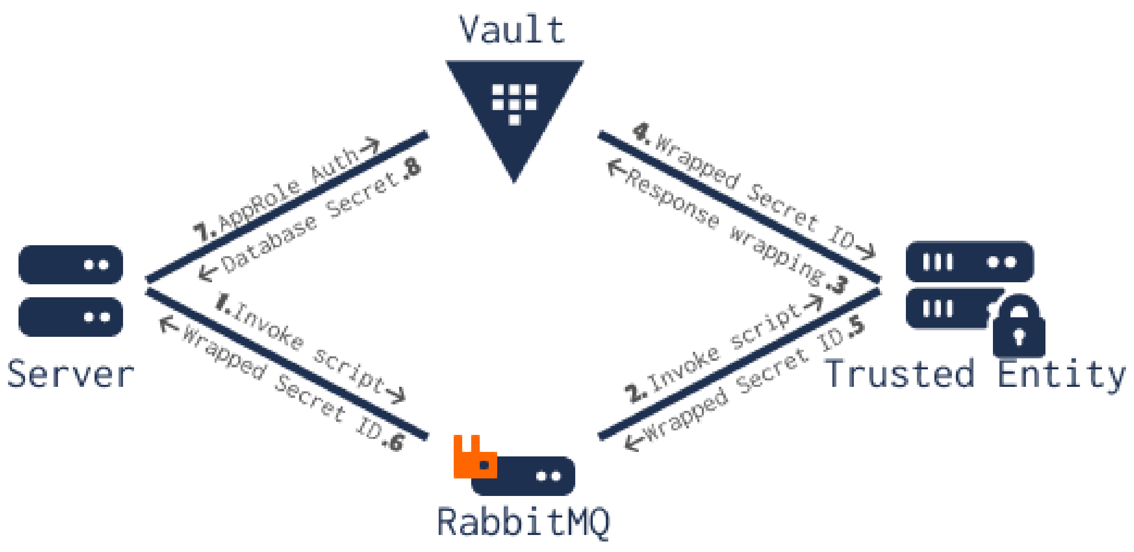
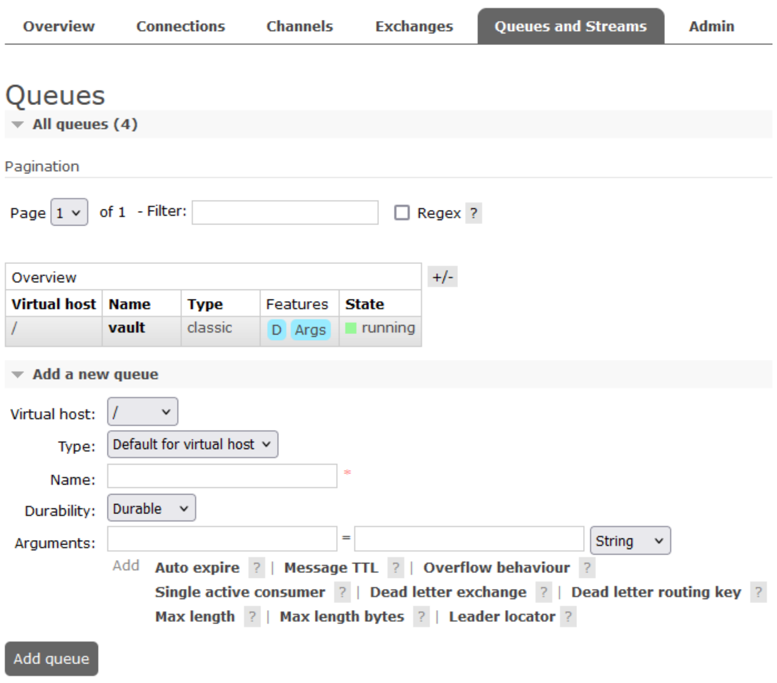


Figure 2.4.1: RPC Workflow

As shown in Figure 2.4.1, we already looked at what's going on with the first step of remotely invoking the script, by analyzing how to set up a channel and publish a message to it with RabbitMQ. We will configure trustedentity to listen to the right queue for incoming messages, but first let's see how we can bind routing keys to an exchange, using the management plugin RabbitMQ offers to access its web interface:



Queues

All queues (4)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview

Virtual host	Name	Type	Features	State
/	vault	classic	D Args	🟩 running

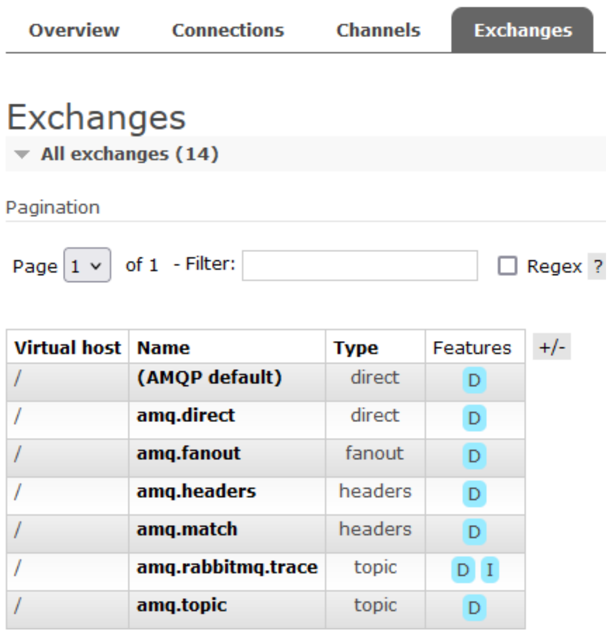
Add a new queue

Virtual host:	/
Type:	Default for virtual host
Name:	*
Durability:	Durable
Arguments:	= String

Auto expire ? | Message TTL ? | Overflow behaviour ?
Single active consumer ? | Dead letter exchange ? | Dead letter routing key ?
Max length ? | Max length bytes ? | Leader locator ?

Figure 2.4.2: Queues, in RabbitMQ management

First, we create the queue, by navigating to the "Queues and Streams" category, as shown in Figure 2.4.2. This is what the routing key will be targeting, which in turn is handled by the exchange. Most of the options while creating a new queue can be left as defaults, but one important distinction needs to be made between the two durability modes, durable and transient. For the main queue that is listening for messages in trustedentity, we will go with a durable queue since those are persistent between container restarts, but for the queues that we declare in server, in Listing 2.4.2 from last page, we create them as transient by setting the 2nd argument to `false`, which means they are lost on container restarts (see RabbitMQ, n.d.-a, for more detail), and this is intentional because each queue that's created is client specific and does not need to live long-term.



Virtual host	Name	Type	Features
/	(AMQP default)	direct	D
/	amq.direct	direct	D
/	amq.fanout	fanout	D
/	amq.headers	headers	D
/	amq.match	headers	D
/	amq.rabbitmq.trace	topic	D I
/	amq.topic	topic	D

Figure 2.4.3: List of exchanges, in RabbitMQ management

By default, RabbitMQ provides various exchange types, as shown in Figure 2.4.3, but the one we are interested in is `amq.direct`. If we navigate into it, we can find a section to add a binding directly to the exchange. The queue we just created can be chosen here, and the routing key can be named anything as long as it is bound to a valid queue and is properly referred to when publishing a message.

With the exchange listing the routing key to the right queue, and with the server container able to publish a message in the channel, we can move onto the consumer, trustedentity.

From the Dockerfile shown in the previous subchapter, Listing 2.3.9, an important file is included in the build at the path `/var/www/html/mq-consume.php` . This file, while supported by Apache, will be the container's foreground process, and it is responsible for listening to incoming messages at the queue we defined through the RabbitMQ interface. It will also respond with confirmation messages, when the procedure it's supposed to carry out is completed, namely executing the script file.

To set `mq-consume.php` as the foreground process, we can set the entrypoint of the trustedentity container in the compose file by running the PHP process, with the file's path as an argument: `entrypoint: ["php", "/var/www/html/mq-consume.php"]`. The actual Response Wrapping script will be called from this PHP file.

```php
// -- Consume from remote --
$channel->basic_consume(
    'vault',  // Queue name
    '',       // Unique consumer tag
    false,    // No local
    true,     // No ack
    false,    // Not exclusive
    false,    // Wait for confirm
    $callback // Callback function
);
// -- End consume from remote --
while ($channel->is_consuming()) $channel->wait();
```
Listing 2.4.3: Consumer listening for messages, in trustedentity, mq-consume.php

In Listing 2.4.3, we can see how simply we can choose the right queue to listen to by passing the name of the queue we set up earlier as the 1st argument in `basic_consume()`. A while loop is also introduced, so that the PHP file can continue running as the foreground process indefinitely as it's listening for messages.

An important bit to mention from the same Listing, is the callback function. This is the function that will be executed in the PHP file once a message is received in the `vault` queue:

```php
$callback = function ($msg) use ($channel) {
    $bodyData = json_decode($msg->body, true);
    $sessionID = $bodyData['session_c_id'];
    exec("/bin/bash ../../../vault/vlt-secure.sh '$sessionID'");
    $response = ['session_c_id'  => $sessionID];
    $response = json_encode($response, true);
    $outgoingMsg = new AMQPMessage($response, ['correlation_id' => $msg-
>get('correlation_id')]);

    // -- Publish to remote --
    $channel->basic_publish(
        $outgoingMsg,         // Msg
        '',                   // Unique consumer tag
        $msg->get('reply_to') // Routing key
    );
    // -- End publish to remote --
};
```
Listing 2.4.4: Callback function, in trustedentity, mq-consume.php

This is almost the end of the RPC's lifecycle! In Listing 2.4.4, the callback function, here a closure, inherits the channel we first opened in `mq-consume.php`, and is passed the message that was received as the only argument. The custom session ID we encoded in the message in the server container is now read locally, and sent into the script file (from Listing 2.3.4, from the last subchapter) with the `exec()` function. Now that the script has been invoked, the file at `/pipe/wsID_$path.txt` is populated with the wrapped Secret ID.

With the wrapped Secret ID in the file, we can say that the procedure is complete, so all that's left is to return a message to the server container to confirm this. Looking at Listing 2.4.4 again, we populate the body of the return message with some value, that way we can satisfy the while loop's condition of a non-`null` value in the server container, and most importantly, include the correlation ID that was used to sign the original message with. The message is then published, using the name of the unique queue we declared in the producer as the routing key. The reason this works, is because when the exchange parameter/argument is left as the empty string in `basic_publish()` (2nd parameter), we choose to publish in the default exchange with the queue name being the routing key (see RabbitMQ, n.d.-b, for more detail). In the server container, this is how it's listening for this message:

```
$channel->basic_consume(
            $callbackQueue, // Queue name
            '',             // Unique consumer tag
            false,          // No local
            true,           // No ack
            false,          // Not exclusive
            false,          // Wait for confirm
            function (AMQPMessage $msg) use (&$response, $correlationId) {
                if ($msg->get('correlation_id') === $correlationId) {
                    $response = $msg->body;
                }
            }
);
```

Listing 2.4.5: Message listener, in server, mq-produce.php

Again seen in Listing 2.4.5, is the same technique with the unique consumer tag, but this time to listen to the default exchange. Now when the message for the queue that was declared previously is received, and the correlation ID matches, `$response` is populated and it breaks the loop that's suspending `mq-produce.php`, and allows server to read the wrapped Secret ID.

3 Secrets in action, with Vault & PDO

With the previous chapter focusing mostly on setting up Vault to fetch secrets, and showcasing a method of securely introducing the Secret Zero to the main Docker container with RabbitMQ, this chapter will be dedicated to bringing all of the API calls that make this possible together, as well as using the secrets we obtain to finally make a connection to the database with PDO.

3.1 API call chain

By the end of the previous chapter we had just seen the Secret Zero written into a file in a shared volume called `pipe`, and just before closing subchapter 2.3 we saw how we can open the file and read the secret in Listing 2.3.11. That particular function happens to be the last in the chain of API calls, which means it’s the perfect one to start from and move backwards from there. The goal is to summarize how all the API calls shown throughout chapter 2 are connected.

For the function itself shown in Listing 2.3.11 (token unwrap API), we have already seen a very similar pattern in the structure in other API calls throughout chapter 2. None of the functions we’ll be creating are closures, or anonymous, so we can simply pass any variables we’ve defined locally as arguments. A common one will be the vault address (`$vA`), which is why it was made an environment variable, so it can be used in any file.

The main part of each function, like the one in Listing 2.3.11, is the Guzzle method to create a client object with `$client = new Client();` which is pretty consistent across the functions we’ll make for our API calls. Like we’ve seen with other functions in chapter 2 and now in this Listing, the HTTP request is made with `$client->post()` or `$client->get()` and its `$response` can be converted into a PHP array with `json_decode()`. The PHP array structure will almost always be different for each API call, so a good way to know which keys we can access is to either use the API testing tool in the Vault web interface, or `echo` the array with this very useful technique:

```
echo '<pre>';
print_r($body);
echo '</pre>';
```
Listing 3.1.1: Viewing an array's structure

What the code in Listing 3.1.1 does is print the array using the preformatted text tag to preserve line breaks (see W3Schools, n.d.-a, for more detail), and is probably the most clear way to print an array with PHP. Do **not** use this in a production environment, as it can expose sensitive information you wouldn't want clients to see.

Now that we have a better understanding of the common parts of an API call function, let's look at an overview of how I decided to separate the backend files from unwrapping Secret Zero to making the connection to the database:

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

From left to right, each file requires the next one in the chain to populate a set of variables in order to execute the function in it, except the last two, where mq-produce requires both vault-connect-2 and vault-key-unwrap. This type of file management might not be necessarily the best way to structure a linear procedure, and there might be some good arguments against separating everything up into multiple files and making it confusing for future developers that will eventually read the code, but I think this is down to preference. Each file clearly indicates what file it requires getting its data from at the very top, making navigation pretty simple, and it's consistent in that each file contains one function/API call, but again that is just preference.

The files in the overview will be color-coded to make it clear what we've already visited, and what we are about to look at.

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

The `vault-key-unwrap.php` file has already been shown in Listing 2.3.11, so we can move on to `vault-connect-2.php`. In chapter 2, we made an RPC using RabbitMQ to send a message from the server container, to trustedentity. Something I omitted from the configuration, is showing how the connection is done to RabbitMQ. In total, we need the address, port, username, password, and virtual host. For the virtual host, we can use the one provided to us by default from RabbitMQ, which is the forward slash `'/'`. The address is the container's name, and the port can be the default AMQP port for RabbitMQ that's used for connections publishing or consuming messages, which is `5672`.

Now, here's the catch: To some degree, we will have to make the username and password available to the server container so that it can connect to RabbitMQ. It doesn't make sense to put the credentials into a second Vault, cause then we'd need a different message broker to RPC to get the credentials of RabbitMQ, and we'd just end up needing access to the first secret to access the rest. Secret Zero all over again. However, remember that the messages RabbitMQ sends don't have any sensitive information tied to them, so even if they were intercepted by a malicious entity, they wouldn't gain any information about the actual Secret Zero, therefore, the credentials to RabbitMQ can either 1) be directly hardcoded into the environment, or 2) be put into the Vault and have a token generated with a policy attached to it that allows access only to that secret, then hardcode the token into the environment. The RabbitMQ user can also have most of its privileges dropped to the absolute least necessary to read or write into specific queues.

The way I introduce the RabbitMQ credentials into the environment, is with the token method, thus why `vault-connect-2.php` is in the chain.

```php
$vaultAddress = getenv('VAULT_ADDRESS');
$vaultToken = getenv('VAULT_TOKEN_SCRT_2');
function getSecret2(string $vA, string $vT) {
    $client = new Client([
        'base_uri' => $vA,
        'headers' => ['X-Vault-Token' => $vT],
    ]);
    try {
        $response = $client->get('/v1/kv/data/mqsecret');
        $body = json_decode($response->getBody(), true);
        return $body['data']['data'];
    } catch (Exception $e) {
        die("Failure: <h2><b>Message:</b></h2> " . $e->getMessage() .
PHP_EOL);
    }
}
```

Listing 3.1.2: Fetching RabbitMQ credentials from Vault, in server, vault-connect-2.php

Again, in Listing 3.1.2, we can see a lot of the common elements and structure in the API call. The only things that are different are the path of the API `/v1/kv/data/mqsecret`, the request method, and the array structure.

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

Moving on to `mq-produce.php`, while remembering we are going in reverse in the requirement order, we now have the unwrapped Secret ID and the RabbitMQ credentials. This node split might be the most confusing to understand, but it gets linear pretty quick after this file.

```php
require 'vault-key-unwrap.php';
require 'vault-connect-2.php';
$data = getSecret2($vaultAddress, $vaultToken);
$mqConfig = [
    'mq_address' => 'mq',
    'mq_username' => $data['MQ_USER_NAME'],
    'mq_password' => $data['MQ_PASSWORD'],
];
function routeMessage(array $mqC) {
    $mqA = $mqC['mq_address'];
    $mqU = $mqC['mq_username'];
    $mqP = $mqC['mq_password'];
    $vaultAddress = getenv('VAULT_ADDRESS');
    $connection = new AMQPStreamConnection($mqA, 5672, $mqU, $mqP, '/');
    $channel = $connection->channel();
...
    $roleSecretID = unwrap($vaultAddress);
...
    return $roleSecretID;

 }
```

Listing 3.1.3: RabbitMQ connection, in server, mq-produce.php

As we can see in Listing 3.1.3, we call the Vault API to fetch the RabbitMQ secrets with `getSecret2()` from `vault-connect-2.php` and then initialize an array with the configuration. This data is passed as an argument into `routeMessage()` which is called from the next file in the chain. This particular function has already been dissected in chapter 2, where we discuss the RPC and was mostly summarized to look at a few things we missed, like the RabbitMQ connection. In this function we also call `unwrap()`, from `vault-key-unwrap.php` (from Listing 2.3.11). This function returns the Secret ID to the next function we'll be discussing.

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

```php
require 'mq-produce.php';
$vaultAddress = getenv('VAULT_ADDRESS');
$roleID = getenv('VAULT_ROLE_ID');
$roleSecretID = routeMessage($mqConfig);
function auth(string $vA, string $rsID, string $rID) {
    $client = new Client([
        'base_uri' => $vA,
    ]);
    try {
        $response = $client->post('/v1/auth/approle/login', [
            'json' => [
                'role_id' => $rID,
                'secret_id' => $rsID
                ]
        ]);
        $body = json_decode($response->getBody(), true);
        return $body['auth']['client_token'];
    } catch (Exception $e) {
        die("Failure: <h2><b>Message:</b></h2> " . $e->getMessage() . 
PHP_EOL);
    }
}
```

Listing 3.1.4: AppRole authentication with Vault, in server, vault-auth.php

The function in Listing 3.1.4 in its completeness (partly shown before in Listing 2.3.12 in subchapter 2.3) shows how the obtained Secret ID is used to authenticate with AppRole, by invoking the `/v1/auth/approle/login` API endpoint. The response we get from Vault if both the role ID and role Secret ID are correct is a token with access to the database secret. By this point, it’s also clear how API calls are chained, by invoking the function from the next file in the chain to assign data in a variable that’ll be the argument for the current function.

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

Next up, we can use the token we obtained with AppRole to access the secret.

```
require 'vault-auth.php';
$vaultAddress = getenv('VAULT_ADDRESS');
$vaultToken = auth($vaultAddress, $roleSecretID, $roleID);
function getSecret(string $vA, string $vT) {
    $client = new Client([
        'base_uri' => $vA,
        'headers' => ['X-Vault-Token' => $vT],
    ]);
    try {
        $response = $client->get('/v1/kv/data/databasesecret');
        $body = json_decode($response->getBody(), true);
        return $body['data']['data'];
    } catch (Exception $e) {
        die("Failure: <h2><b>Message:</b></h2> " . $e->getMessage() .
PHP_EOL);
    }
}
```

Listing 3.1.5: Fetching the database secret, in server, vault-connect.php

The function shown in Listing 3.1.5 is the last Vault API call in the chain, with the actual secret being fetched by invoking the endpoint `/v1/kv/data/databasesecret`. **Note** that for all of these functions where the flow is suspended if there is an error reaching the API endpoint with `die()`, the choice there is intentional and should be done with great care not to expose any sensitive information. The best way to handle errors in a production environment is to instead log the error in a file with the timestamp and endpoint, and redirect the client while handling the error programmatically.

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

Having finally obtained the secret, we need to separate the data to get the username, password, and optionally the database name if we didn't hardcode it in the environment. Any other database specific configuration is defined here, like the hostname, port, connection timeout, etc.

```
require 'vault-connect.php';
$data = getSecret($vaultAddress, $vaultToken);
$password = $data['PG_PASSWORD'];
$username = $data['PG_USER_NAME'];
$databasename = $data['PG_DATABASE_NAME'];
$hostname = getenv('PG_HOST_NAME');
```

Listing 3.1.6: Preparing database configuration, in server, db-config.php

In Listing 3.1.6 we access the keys in the database secret with the names we assigned them with originally in the interface, in chapter 2, subchapter 2.2.

db-connect -> db-config -> vault-connect -> vault-auth -> mq-produce -> (vault-connect-2, vault-key-unwrap)

In `db-connect.php`, we will use the data we gathered in our configuration to make the connection to the database.

```
require 'db-config.php';
try {
    $dsn = "pgsql:host=$hostname;dbname=$databasename;";
    $pdo = new PDO($dsn, $username, $password, [
        PDO::ATTR_PERSISTENT => TRUE,
        PDO::ATTR_TIMEOUT => 5,
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
    ]);
} catch (PDOException $e) {
    die("Failure: <h2><b>Message:</b></h2> " . $e->getMessage() . PHP_EOL);
}
return $pdo;
```

Listing 3.1.7: Making a connection to the database using PDO, in server, db-connect.php

As shown in Listing 3.1.7, the connection script in its completeness (partly shown before in Listing 1.4.2, in subchapter 1.4). More detail on exactly what a PDO connection is and why it's used instead of a normal connection, in the next subchapter.

Each file that now requires `db-connect.php` can utilize the database connection. The only thing left to discuss is the session-based variable used to set the name of the file Secret Zero gets written into. For this, we can require another file that contains just the session logic, called `session.php`.

```
session_start();
if (!isset($_SESSION['session_c_id'])) {
    $_SESSION['session_c_id'] = generateRandomString();
}
function generateRandomString($length = 10) {
    return
substr(str_shuffle(str_repeat($x='0123456789abcdefghijklmnopqrstuvwxyzABCDE
FGHIJKLMNOPQRSTUVWXYZ', ceil($length/strlen($x)) )),1,$length);
}
```

Listing 3.1.8: Setting up a session, in server, session.php

A session needs to be started for each client that needs a database connection. In Listing 3.1.8 we can see how the custom session ID is generated for the purpose mentioned earlier. In particular, the `generateRandomString()` function is obtained from A. Cheshirov (2012) and generates a string that's unique enough (62^10 = 839 quadrillion possibilities) for a session-based use case.

3.2 PostgreSQL interaction with PHP PDO

Now that we have the entire API call chain ready, we can make a PDO connection to PostgreSQL. Choosing PDO to connect to the database with is done because we can utilize prepared statements and bound parameters, which take care of a lot of the sanitation required each time we deal with user inputs in particular. This ultimately protects against tampering like SQL injection, where the user maliciously enters a string that's designed to escape the input's intended use case, and execute an entirely different query. The way PDO protects against this, is by first sending the actual statement template to the database, and then transmitting the values with a different protocol that's distinct from the SQL syntax (see W3Schools, n.d.-b, for more detail). The database can then combine these two ensuring that the values are not treated as executable code.

Using the code shown in Listing 3.1.7, we return the `$pdo` object back to the PHP file that required `db-connect.php`. We can use that object to prepare statements with, and execute queries on. To test how this works, and the connection itself, we can log in to adminer, the web interface that we set up in chapter 1, subchapter 1.3, to access the database and create a temporary table.

At first, when we log in, we can see that by default there is a schema already created for us, called public (Figure 3.2.1). This particular schema allows all users of the database to access and create (only if upgrading from PostgreSQL 14 and earlier) objects within it (see PostgreSQL, n.d-a, for more detail). If the idea of a schema sounds confusing, it's understandable. For example, MySQL doesn't have "schemas" in a database, and instead we'd say that the database itself is the schema. For PostgreSQL however, a schema acts like a namespace, in that it organizes objects together. For instance, were for object names to be conflicting (e.g. two tables with the same name), they could be separated into different schemas and be accepted.

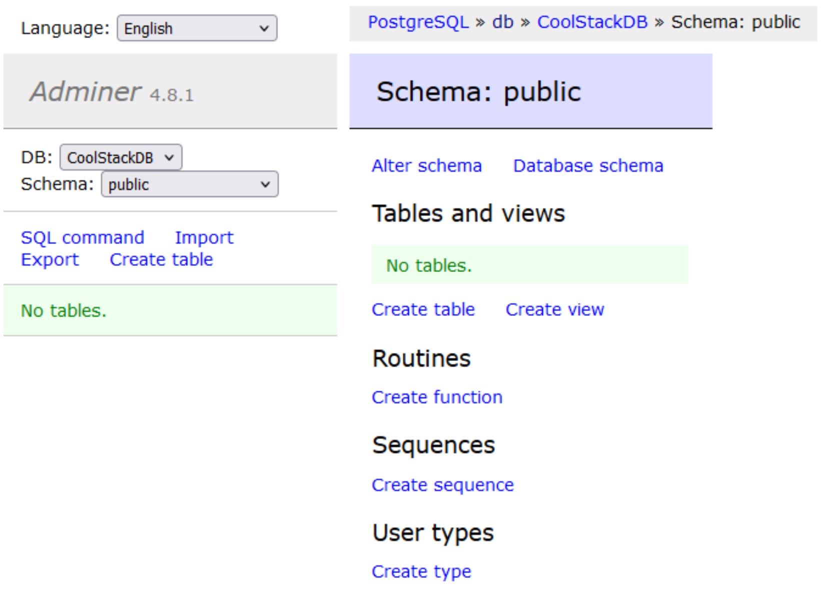
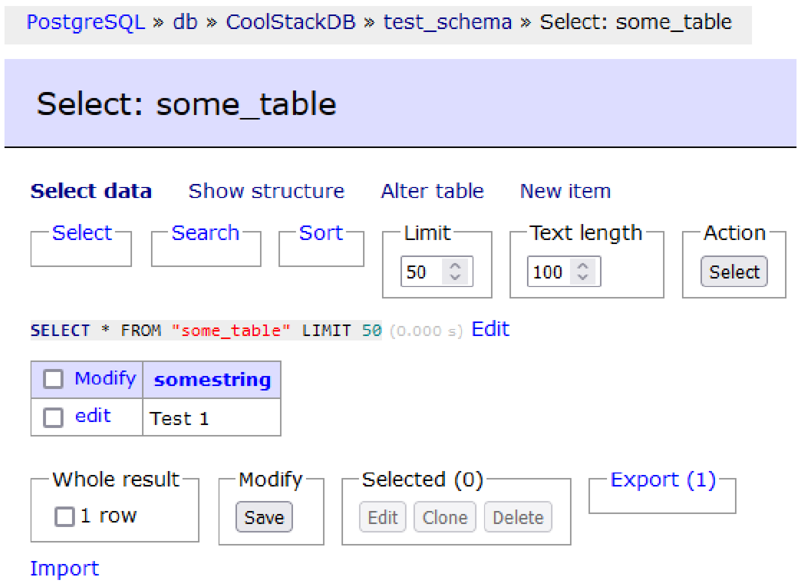


Figure 3.2.1: Adminer web interface, logged in view

If we are interested in creating multiple users and multiple schemas, where each user has limited privileges to each schema, dropping the public schema and starting with a new one makes the most sense. For the example ahead, as well as in the next chapter that goes more in-depth with the database setup, we will be dropping the public schema and creating two new ones, with the user `POSTGRES_USER` created during the Docker compose stage. Note that this user is by default a superuser, meaning it has permissions to create, alter, and drop any database objects, including the database we specified in the environment with `POSTGRES_DB` but not limited to it.

The interface has an SQL command page, in case the developer is interested in executing queries for testing purposes, before involving PHP/PDO in any way.

In the example ahead, a new schema is created called `test_schema`, with a single table called `some_table`, with a field called `somestring`. We also create a row for the field so that we can fetch it with PHP. The test values can be seen in Figure 3.2.2.


PostgreSQL » db » CoolStackDB » test_schema » Select: some_table

Select: some_table

Limit: 50 · Text length: 100

```
SELECT * FROM "some_table" LIMIT 50 (0.000 s) Edit
```

	somestring
edit	Test 1

Figure 3.2.2: Test values on the database, in adminer interface

Since for this example we dropped the public schema, we will need to change the search path from pointing to the public schema (by default) to `test_schema`. The search path is a run-time parameter, and it is used to control the schema search order. If only one schema is defined in the order, then essentially it's like telling PostgreSQL to look in that specific schema to reference database objects. If we have more than one schema and we frequently use them all, then we can make a function that sets the path to each specific schema, and use the corresponding function before executing statements to that schema.

```
function setDbSearchPathTestSchema($conn) {
    $query = "SET search_path TO test_schema";
    $statement = $conn->prepare($query);
    $statement->execute();
    return;
}
```

Listing 3.2.1: Setting the search path for PostgreSQL, in server, index.php

The function we define in Listing 3.2.1 can be called right before any `test_schema` queries are sent to the database, where `$conn` is the PDO object we create with the database connection at first.

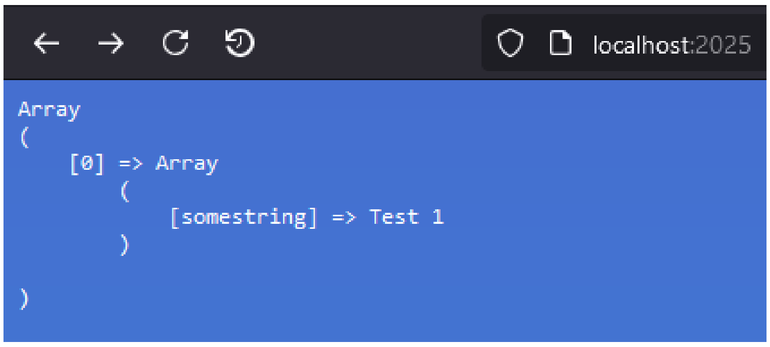
Finally, to see that our connection works, we can fetch the row that we created in `some_table`.

```php
$query = "SELECT somestring
          FROM some_table";
$statement = $conn->prepare($query);
$statement->execute();
$result = $statement->fetchAll(PDO::FETCH_ASSOC);
echo '<pre>';
print_r($result);
echo '</pre>';
```

Listing 3.2.2: PostgreSQL statement, in server, index.php

A very simple query, with a very critical outcome! At this point, if our HTML is printed at all when navigating to `localhost:2025`, it means we have no fatal errors with our PHP code and that we should be able to see the array returned by `fetchAll()`. It is exciting to realize that not only does the code from Listing 3.2.2 work, not even just that the database connection is working, but the entire backend that's responsible for delivering the credentials to make that connection in the first place.

Getting back on track, it is to be noted that `fetchAll()` exhausts the rows returned by `$statement`, so if we want to fetch any more rows we'll have to make a new statement to the database.



```
Array
(
    [0] => Array
        (
            [somestring] => Test 1
        )

)
```

Figure 3.2.3: Result of the database statement, in server

A successful outcome can be seen in Figure 3.2.3, with an array having the structure we expected. If we had more rows in `some_table`, those would all be fetched with the query we executed in Listing 3.2.2.

The server is now ready to make statements to the database. Next up, is creating an admin page to manage data on the website we are serving on the server container.

4 Creating a Custom CMS

An efficient way to manage the content of our website, whether that is the header of a paragraph, the paragraph itself, a video that's supposed to play in a particular section, or the names of the website's categories, is to separate the manual need of querying the database and provide tools that the website owner can use to communicate with it instead at a high level. There are already a lot of tools like that, with popular choices like WordPress and Joomla, but I was curious enough to create my own.

The functions I have prepared for this chapter mostly revolve around creating and managing categories, as well as the content the categories can represent. For example, a category might have a title and excerpt that are form-fitting on the front page, but also an inner content page, with a longer title and a paragraph fully describing the category.

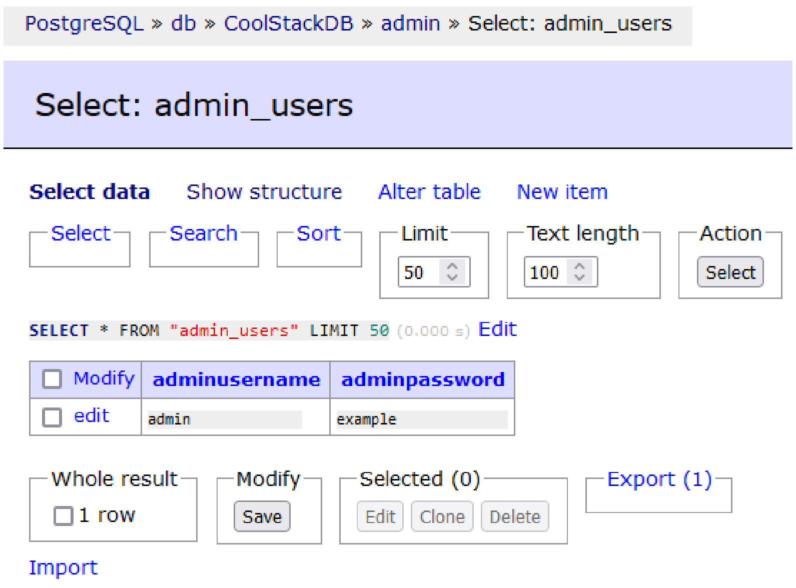
4.1 Account login & CSRF prevention

By introducing an admin page, we are also creating the need for a responsible way to permit access to the very important functions that the page itself encapsulates. One of the most immediate subjects worth discussing regarding this is locking the admin space down by requiring an account. Of course, this in and of itself can introduce vulnerabilities as well as open itself up to brute-force attacks.

Along with building an account login flow, we will also discuss how Cross-Site Request Forgery (CSRF) attacks can be prevented, and what the implementation of that means for the login flow. This type of attack is done by leveraging authenticated user sessions to perform unauthorized actions on their behalf (see Owasp, n.d.-a, for more detail). For instance, an attacker could disguise a form, the forged request, as an irrelevant action in a completely different website to ours, and so long as a trusted user has already authenticated through the admin login page, performing that misleading action on the remote website could result in the user's saved login session to be used by the attacker to execute actions to our website without the user's knowledge.

Coincidentally, the same technique we'll use to prevent CSRF attacks will also ensure that only authorized requests will be able to interact with the endpoint corresponding to the login handler. The way we authorize those requests will become clear once we implement the technique, but of course we'll need a login flow first before we can do that.

For the admin page it makes the most sense that users can't create new accounts, but can only login using credentials provided to them by an administrator, and as such we'll assume that the account creation procedure is completely separate from the admin page. For the example ahead, an account is manually created in adminer, in a new schema called `admin`, in a table called `admin_users` (Figure 4.1.1).



```
SELECT * FROM "admin_users" LIMIT 50 (0.000 s) Edit
```

	adminusername	adminpassword
Modify	adminusername	adminpassword
edit	admin	example

Figure 4.1.1: Admin page credentials, in adminer

The admin page itself will be separate from `index.php`, and as such can be a new file, `admin.php`. For the login function, we can use a form with two inputs and a submit button. Additionally, we can include a hidden input that's simply used to supply information about the type of form that is sending the data. Even for a simple group of elements such as this, I like to deal with it in parts: The frontend, and the backend component, always starting with the former of the two.

```
...
<form class="login-form" action="admin-functions.php" method="POST">
   <label for="uname" hidden></label>
   <input class="login-field" type="text" id="uname" name="uname"
placeholder="Username..." required maxlength="18">
   <label for="pword" hidden></label>
   <input class="login-field" type="password" id="pword" name="pword"
placeholder="Password..." required maxlength="20">
   <input type="hidden" id="flow-action" name="flow-action" value="login">
   <input class="login-submit font-text1 color-r1" type="submit"
value="Sign in">
</form>
...
```

Listing 4.1.1: Frontend component of login function, in server, admin.php

Seen in Listing 4.1.1 is purely the frontend/HTML component of the login function (rendered in browser in Figure 4.1.2). Based on the form's configuration, the values of the three inputs inside the form – one for the username, one for the password, and another for the form information – upon being submitted, will be sent to a file called `admin-functions.php` with the POST method. If you remember the way we invoked Vault API endpoints in chapter 3, the form's hidden input `flow-action` might feel familiar if it was described as the endpoint, and the file that is getting called, `admin-functions.php`, the API. Ultimately, the value of the hidden input decides which function will be executed in that file.

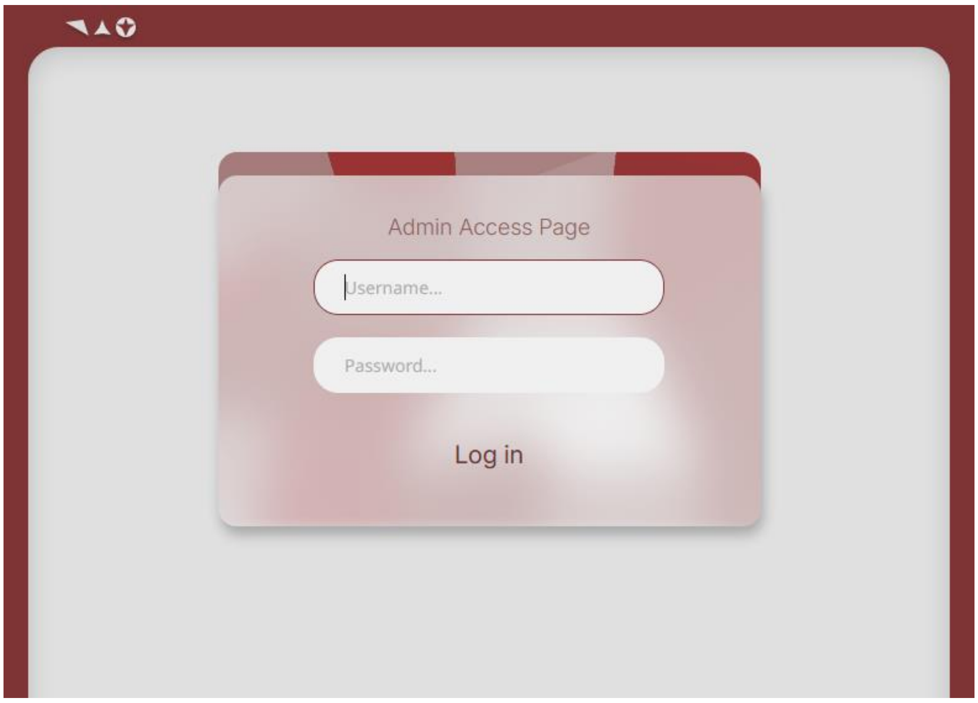


Figure 4.1.2: Login view, in server, admin.php

With the frontend component taken care of, let's take a look at how `admin-functions.php` handles data it receives by a form submitted from `admin.php`, with the value of the `flow-action` input set to `login`:

```
require 'db-functions.php';
$conn = $pdo;
$action = $_POST["flow-action"];
if (isset($action)) {
    switch ($action) {
        case "login":
            $userSelectedName = $_POST["uname"];
            $userSelectedPass = $_POST["pword"];
            $dbResult = getAdminCreds($conn, $userSelectedName);
            if (!empty($dbResult)) {
                $dbReturnedPass = trim($dbResult[0]['adminpassword']);
                if (hash_equals($dbReturnedPass, $userSelectedPass)) {
                    // everything was right!
                    session_regenerate_id();
                    $_SESSION['session_admin_access'] = true;
                    $_SESSION['session_current_admin_user'] =
                    $userSelectedName;
                    header("Location: admin.php");
                    exit;
                } else {
                    // wrong password!
                    $_SESSION['session_admin_access'] = false;
                    header("Location: admin.php");
                    exit;
                }
            } else {
                // wrong username!
                $_SESSION['session_admin_access'] = false;
                header("Location: admin.php");
                exit;
            }
...
} else header("Location: admin.php");
```

Listing 4.1.2: Login handler, in server, admin-functions.php

In Listing 4.1.2, we can see one way a login procedure can be realized, using a session. First of all, we depend on the API chain we created in chapter 3 to validate user input against the database. `db-functions.php` satisfies this dependency requirement, as well as provide the space we will be declaring all of the database specific functions.

In order to build a scalable structure for `admin-functions.php` we can make use of a switch statement, which will decide which function is going to be executed based on the type of form that sent the data, and that's nothing more than the value of `flow-action`.

To access the value of any input data sent by the form, we can use the superglobal `$_POST`. Capturing the value of `flow-action` returns the decision for which function will be executed, while `uname` and `pword` return the username and the password the user provided. We can use these values to compare against a valid user with matching credentials in the database. Such a function like mentioned earlier will find its home in `db-functions.php`.

```
require __DIR__ . '/session.php';
require __DIR__ . '/db-connect.php';
function setDbSearchPathAdmin($c) {
    $query = "SET search_path TO admin";
    $statement = $c->prepare($query);
    $statement->execute();
    return;
}
function getAdminCreds($c, $uSN) {
    setDbSearchPathAdmin($c);
    $query = "SELECT adminusername, adminpassword
              FROM admin_users
              WHERE adminusername = :uSN";
    $statement = $c->prepare($query);
    $statement->bindParam(':uSN', $uSN);
    $statement->execute();
    $result = $statement->fetchAll(PDO::FETCH_ASSOC);
    return $result;
}
...
```

Listing 4.1.3: getAdminCreds() function, in server, db-functions.php

In Listing 4.1.3, we show the relevant function, `getAdminCreds()`, as it's declared in `db-functions.php`. It uses a connection object (`$c`), and prepared statements with bound parameters to query the database. In this particular query, for example, the bound parameter is `:uSN`, which is then sent to the database as raw data separate from the rest of the statement, with `$statement->bindParam(':uSN', $uSN)`. All this function does, is return a username and password pair from the database, based solely on the username provided by the user.

Going back to Listing 4.1.2, we perform a comparison between the user provided password and the one we fetched from the database. The comparison of the password is done here, with PHP, instead of when querying the database intentionally. There is a specific function we can use called `hash_equals()`, that checks if two strings, `known` and `user`, match, but does so by not leaking information about the `known` string. More technically, the function does not return `false` when the first non-matching character is found between the two strings, but instead parses the entire `user` string before doing so. This method is used to prevent timing attacks, where a malicious user can figure out the `known` string by comparing the time discrepancy in execution times between incorrect and correct characters in the `user` string (see Php, n.d.-a, for more detail). In our example, the `known` string is the database fetched password, and the `user` string is the password provided by the user through the form.

Additionally to the timing attack prevention and upon successful login (where both username and password are correct) we can regenerate the session ID assigned when the session was created, to defend against session fixation. This second attack hinges on a user attempting to login to the admin page after having their session identifier baked in the URL with `PHPSESSID` by the attacker (see Shiflett, 2004, for more detail). We can simply regenerate the session ID with `session_regenerate_id()` right before elevating the user's privilege level with `$_SESSION['session_admin_access'] = true`.

In either case where the user has had their privilege level elevated or not, we redirect the flow back to `admin.php` where we can handle the response to the user accordingly, based on this new session variable we set. Consider treating the case where the username or the password is wrong as the same. There is a fine line between user experience and system security, and as far as the experience coming down to simply not being told which of the two credentials was wrong, I'd take my chances and go with the extra security that ambiguity provides here.

As a small note, in Listing 4.1.2, `trim()` is used on the fetched username to remove the database imposed string length. This is specific to using the `character` data type instead of `character varying` in the database, which requires us to specify the length of the string of a field.

Lastly, to use this logged-in state, let’s look at Listing 4.1.4, where the response from `admin-functions.php` is handled. The idea is to build a view based on what the user’s logged-in state is, and for this we can use a switch statement.

```
...
if (!isset($_SESSION['session_admin_access'])) {
    $buildHtml = 1;
    $loginFailure = "";
} elseif ($_SESSION['session_admin_access'] == false) {
    $buildHtml = 1;
    $loginFailure = "Account not found, or user/pass combination is
incorect";
    unset($_SESSION['session_admin_access']);
} elseif ($_SESSION['session_admin_access'] == true) {
    $buildHtml = 2;
    $currentAdminUser = $_SESSION['session_current_admin_user'];
}
...
switch ($buildHtml) {
    // Login screen / Failed login attempt
    case 1:
      ...
      break;
    // Logged-in framework
    case 2:
...
```

Listing 4.1.4: Handling logged-in state, in server, admin.php

Based on whether `$_SESSION['session_admin_access']` is `false` or `true`, we set a variable called `$buildHtml` to 1 or 2, limiting us to the choices of either rendering the regular login screen with the form from Listing 4.1.1, or rendering the logged-in view which we have yet to define. The case where the user has not tried to log in already is also covered with the first case, but missing the information of the failed login attempt.

So far, this is the process for a user to log in, sans the CSRF prevention. To implement one layer of defense is actually fairly simple: Using a CSRF token.

```
$csrfToken = $_SESSION['session_csrf_token'] = generateCSRFToken();
...
$htmlBlock .= '<input type="hidden" id="csrf-token" name="csrf-token"
value="' . $csrfToken . '">';
...
```

Listing 4.1.5: Implementing CSRF protection with unique token, in server, admin.php

In Listing 4.1.5, we include an additional hidden input in the same form we showed earlier, which will bear a unique token generated on every page refresh. This generated token is both used as data in the form, as well as saved as a session variable for comparison later. To capture this new value on the target file (`admin-functions.php`) we can use the `$_POST` superglobal as shown in Listing 4.1.6.

```
...
$action    = $_POST["flow-action"];
$csrfToken =  $_POST["csrf-token"];
if (isset($action) && hash_equals($_SESSION['session_csrf_token'],
$csrfToken)) {
    switch ($action) {
...
```

Listing 4.1.6: Authorizing function access with CSRF token, in server, admin-functions.php

Here, we couple the timing attack defense with the CSRF prevention, using `hash_equals()` to compare the CSRF we just captured from the form's data to the session variable we saved earlier. If these two match, then it means the user is accessing the `admin-functions.php` file with proper authorization.

As for the function that generates this unique token, we can use `random_bytes(32)` for cryptographically secure, "uniformly selected random bytes" (see Php, n.d.-b, for more detail) which can then be converted into a readable format with `bin2hex()`.

Note that the login function won't be the only one using the CSRF token to authorize access. Any form that's handling sensitive information can benefit from this, even those that require a logged-in state, as CSRF attacks mainly target authenticated clients.

4.2 Setting up PosgreSQL tables

The procedure we are about to discuss is one that technically, if done correctly, will only need to be done once, and much like the Vault setup that we saved using the snapshot feature in chapter 2, we will similarly treat our database container with the schema structure by exporting a backup.

I say “done correctly” very carefully because if we assume that the database structure will be used across many applications of ours in the future, the weight of that truth is quite heavy. In reality, there are many things we need to keep in mind, meaning that there are many things we can get wrong. It’ll be common when designing a system that we come up with a new idea, changing the result of the structure ever so slightly. This is what design discovery is, and for our particular application the only way to figure out exactly what the application is supposed to look like is to try stuff out until it gets closer and closer to taking its final shape. The reason I open this subchapter in this way, is to say that I don’t have the best insight in what that final shape is for this application, and that is very important for database structures in particular. The structure that will be shown reflects the shape in so far as I have thought out the use case of the admin page, but there is still a lot of work to be done, and so, let this set the stage for the information that follows.

So far, the database schema for our application consists of a single table that holds the credentials for the admin user. An immediate necessity in the admin page, might be dividing up the actions an administrator can carry out into categories. These can be data in a new table, called `admin_categories`, where each row can have a name and is identified by an ID (Figure 4.2.1). The name will be used to describe the utility of the category, and the ID will be what other tables in the schema can reference in case we need external information to be linked with `admin_categories`, and as such will be a Primary Key.

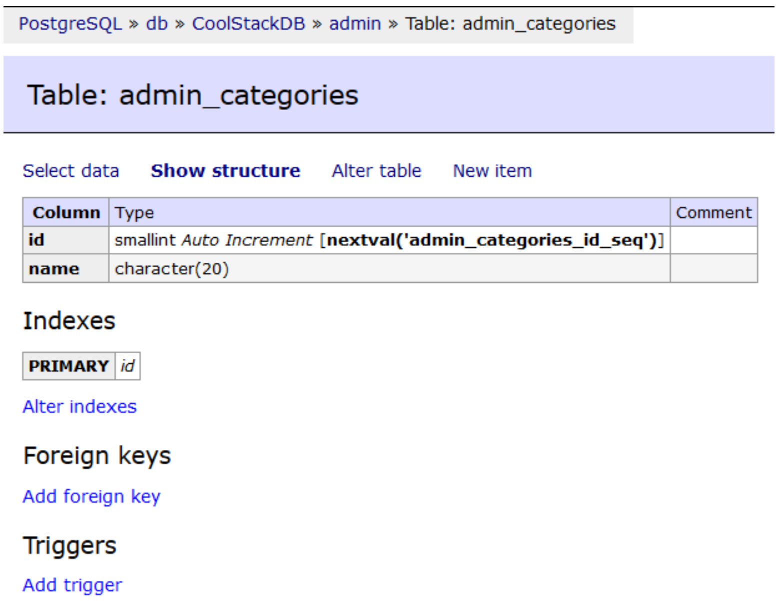

PostgreSQL » db » CoolStackDB » admin » Table: admin_categories

Table: admin_categories

Select data   Show structure   Alter table   New item

Column	Type	Comment
id	smallint *Auto Increment* [nextval('admin_categories_id_seq')]	
name	character(20)	

Indexes

PRIMARY	*id*

Alter indexes

Foreign keys

Add foreign key

Triggers

Add trigger

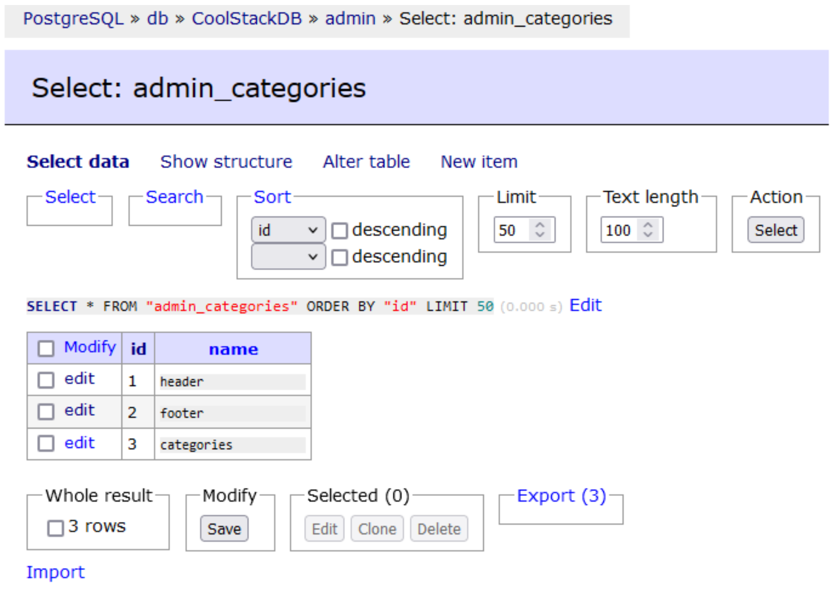
Figure 4.2.1: admin_categories table structure, in adminer, admin schema

This chapter will cover three particular admin page categories (also seen in Figure 4.2.2):

- Categories
- Header
- Footer

Before explaining each one, let's understand the difference between admin categories, and site categories, as the naming is unfortunately similar. Each admin category will be able to govern many site categories, but not the other way around. Each site category is not a row of admin_categories but instead may reference that table by ID. More detail on site categories and what the table for that will look like soon.

The first category type, "Categories", will provide the list of site categories available, as well as the means to edit their attributes or information about them individually. The second, "Header", will be the list of site categories that we want included in the header of the main website, which is traditionally located at the top of the page. Lastly, "Footer", same in functionally as the Header category, but instead will manage the list of site categories we want included in the footer of the main website, located at the bottom of the page.


PostgreSQL » db » CoolStackDB » admin » Select: admin_categories

Select: admin_categories

Select data | Show structure | Alter table | New item

Select | Search | Sort: id, descending | Limit: 50 | Text length: 100 | Action: Select

```
SELECT * FROM "admin_categories" ORDER BY "id" LIMIT 50 (0.000 s) Edit
```

Modify	id	name
edit	1	header
edit	2	footer
edit	3	categories

Whole result: 3 rows | Modify: Save | Selected (0): Edit, Clone, Delete | Export (3)

Import

Figure 4.2.2: Admin category rows, in adminer, admin schema

With the admin categories set up, let’s look at site categories. Since these two category groups represent data about two different spaces, I thought it was best to separate them into different schemas. In a new schema called `site`, a new table is created called `site_categories` (Figure 4.2.3).

Each site category will have columns that describe its attributes. For instance, a site category will surely have a name, and maybe we want to spotlight a few facts about it in a particular section somewhere in the front page, so it’ll have a title and a short description. We can also navigate into this category, either by clicking on it in the header/footer, or by a CTA (call-to-action) button in the spotlight. This page will need a link, and so the site category can also have a permalink that perhaps defaults to whatever the name of the category is, but could be changed to something else entirely. For the inner-page content of the category, maybe we want to include a larger bit of information compared to the shorter, more concise text in the front page, as well as a longer, more fleshed out title to accompany it.

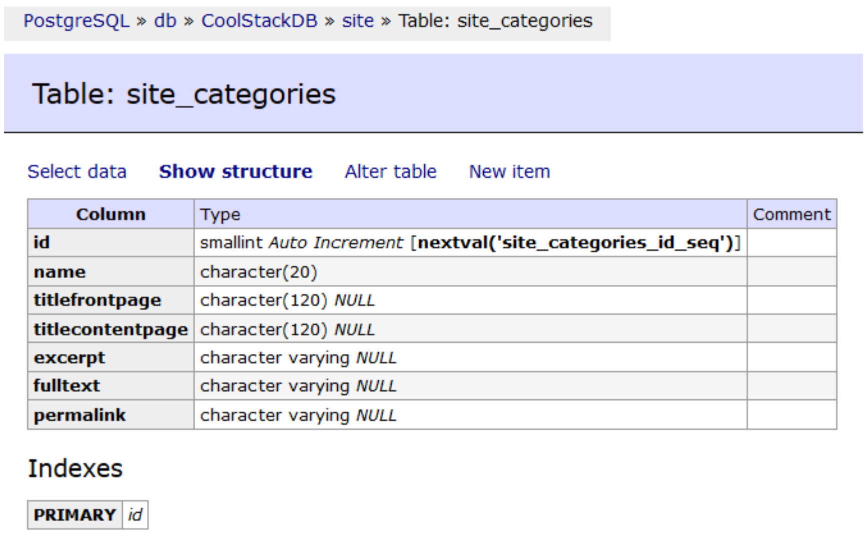

PostgreSQL » db » CoolStackDB » site » Table: site_categories
Select data
Show structure
Alter table
New item

Table: site_categories

Column	Type	Comment
id	smallint *Auto Increment* [**nextval('site_categories_id_seq')**]	
name	character(20)	
titlefrontpage	character(120) *NULL*	
titlecontentpage	character(120) *NULL*	
excerpt	character varying *NULL*	
fulltext	character varying *NULL*	
permalink	character varying *NULL*	

Indexes

PRIMARY	*id*

Figure 4.2.3: site_categories table structure, in adminer, site schema

All of the columns we mentioned previously, minus the name and the permalink that will automatically get the category’s name, are optional during a new category’s creation.

With the admin and site category tables defined, we can now create an association between the two. Specifically, each site category will have a distinct relationship with each admin category. This decision is made because we might want, for example, fewer categories to be shown in the header as opposed to the footer (Figure 4.2.4). The only exception will be with “Categories”, where it’ll include the entire list of site categories.

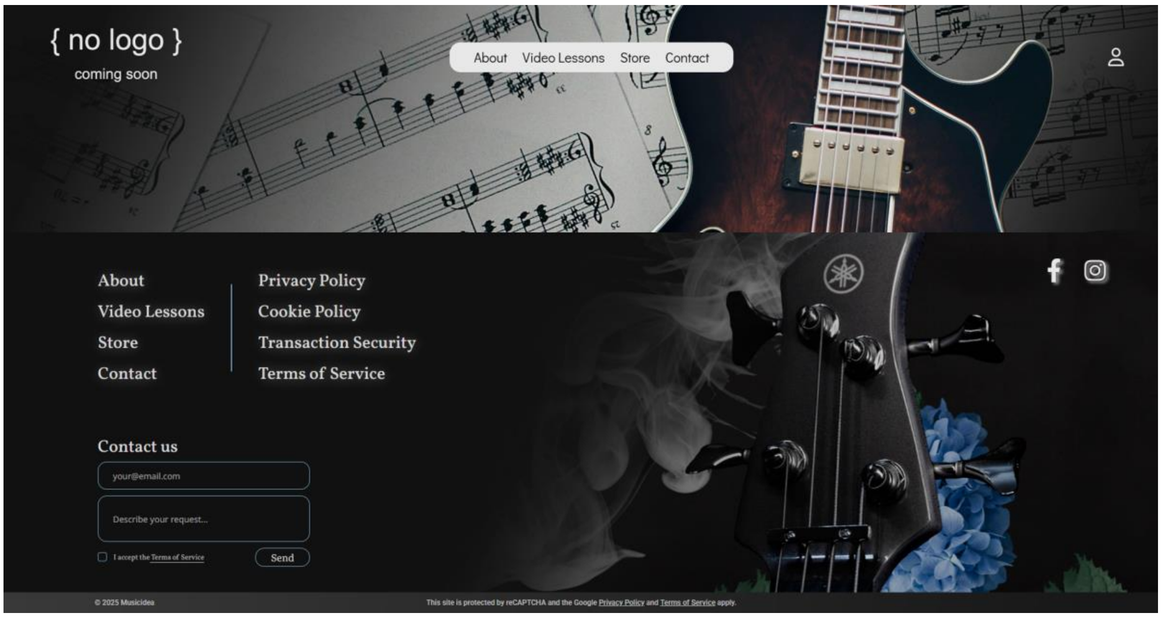


Figure 4.2.4: Example of header vs. footer categories

To achieve this, we need another table that joins `admin_categories` and `site_categories`. Since both of these tables have an `id` field, we can let that be the Foreign Key in the new table, which we can name `category_statuses` (table structure shown in Figure 4.2.5). Here, we can associate each site category (with its `id` field) with an admin category (again, with its `id` field) and specify certain things about that relationship. Some examples are, will that site category be visible in that admin category, or, what is the order of that site category in that admin category?

The status of each site category in each admin category makes the most sense when we think of “Header” and “Footer” categories, but less so for “Categories”. It’s clear that for example, a site category called “Store” might have a visible status for the “Header” category, but be invisible for “Footer”, or vise-versa. All we are saying there, is that we want to see “Store” in the header, but not in the footer of the website. “Categories” has more of an integral role in the admin page compared to the other two where they are more relatable to the main website. For example, “Categories” doesn’t use the visibility attribute because we don’t need to hide site categories from it, but might use the ordering attribute to display the list of site categories in the order those were first introduced to the database. In the end, the biggest use “Categories” gets from this association, is benefitting from the joining of the two tables, `site_categories` and `admin_categories`.

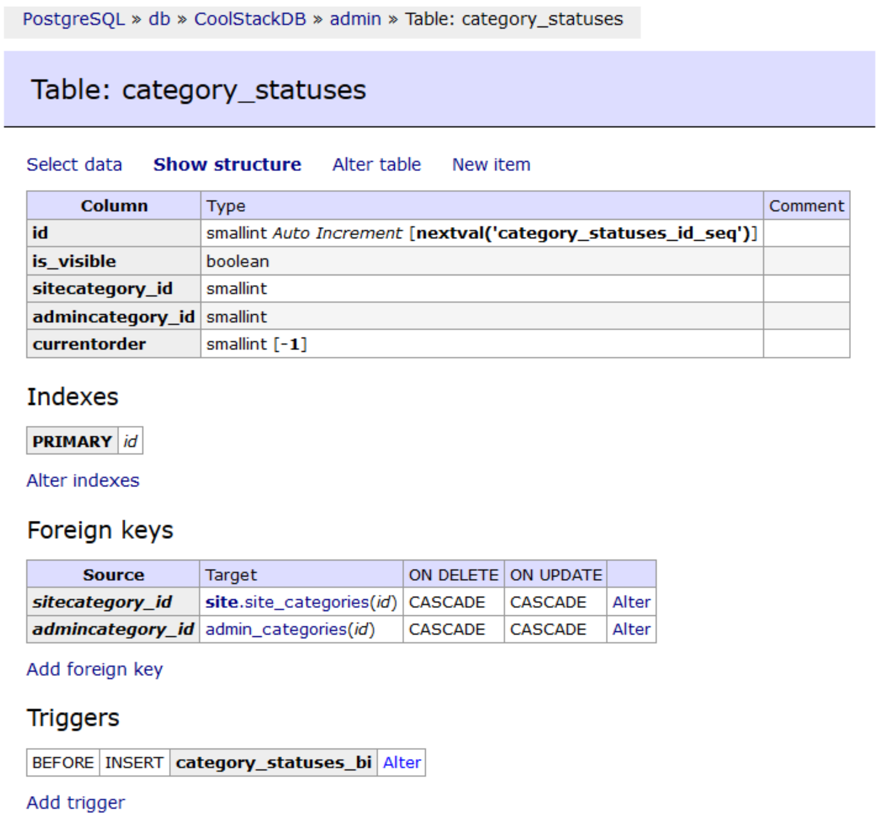

PostgreSQL » db » CoolStackDB » admin » Table: category_statuses

Table: category_statuses

Select data   **Show structure**   Alter table   New item

Column	Type	Comment
id	smallint *Auto Increment* [**nextval('category_statuses_id_seq')**]	
is_visible	boolean	
sitecategory_id	smallint	
admincategory_id	smallint	
currentorder	smallint [**-1**]	

Indexes

PRIMARY	*id*

Alter indexes

Foreign keys

Source	Target	ON DELETE	ON UPDATE	
sitecategory_id	**site**.site_categories(*id*)	CASCADE	CASCADE	Alter
admincategory_id	admin_categories(*id*)	CASCADE	CASCADE	Alter

Add foreign key

Triggers

BEFORE	INSERT	**category_statuses_bi**	Alter

Add trigger

Figure 4.2.5: category_statuses table, in adminer, admin schema

Those are all the tables needed for the admin page functions we will be getting to see in the next subchapter. If we are interested in reusing this schema configuration, we can export the database in adminer at this stage.

4.3 Database & Admin functions

The last time we left off with PHP, was right after having created the login flow for the admin page. Now that we have all of the necessary tables defined in our PostgreSQL database, we can start designing the view after a successful login. Remember, the aim originally was to create different page views depending on the logged-in state, so in the end we should have a single file to render the admin page with. The same principle will be followed each time we want to create a new view for each admin category. An easy way to keep that consistent, is to create a new case in the switch statement per admin category, where the category's name defines a unique case. This is scalable, and it's simple to follow. In the end, we should still have a single file.

Let's start with the default view, where the user is logged in and no categories are selected. For this view, we are mainly going to need to fetch all of the admin categories, and as a bonus we can implement a logout button. For the latter, it's not a database or admin function, so it'll be implemented in `admin.php`.

```
...
if (isset($_GET['logout'])) {
    unset($_SESSION['session_admin_access']);
    unset($_SESSION['session_current_admin_user']);
}
...
<a href="admin.php?logout=true">Log out</a>
...
```

Listing 4.3.1: Logout button, in server, admin.php

As shown in Listing 4.3.1, we can let the button be a link that targets `admin.php`, and pass a parameter that we can then capture as we are loading the same file again and dismantle the login session by unsetting the relevant session variable.

Now for the admin category fetching function. This is purely involving interacting with the database, and as such will be implemented in `db-functions.php`. Paired with `setDbSearchPathAdmin()` to change search path, as shown previously in Listing 4.1.3, we get the function shown in Listing 4.3.2 that fetches all admin categories from the admin schema.

```
function getAdminColumns($c) {
    setDbSearchPathAdmin($c);
    $query = "SELECT name
              FROM admin_categories";
    $statement = $c->prepare($query);
    $statement->execute();
    $result = $statement->fetchAll(PDO::FETCH_ASSOC);
    return $result;
}
```

Listing 4.3.2: Database function to fetch all admin categories, in server, db-functions.php

We can iterate through `$result` with a for loop to get all of the categories, but in and of itself their names alone are not useful. To expand on this, we can build links for each category, using their names, that'll allow us to create an inner page for each one.

```
...
switch ($buildHtml) {
    // Login screen / Failed login attempt
    case 1:
    ...
    // Logged-in framework
    case 2:
    ...
        $dbResult = getAdminColumns($conn);
        foreach ($dbResult as $item) {
            $htmlBlock .= '<a href="admin.php?category=
            ' . $item['name'] . '">' . ucfirst($item['name']) . '</a>';
        }
...
```

Listing 4.3.3: Fetching admin categories, in server, admin.php

As shown in Listing 4.3.3, each category will be a link that targets the file we are already in, `admin.php`, with the parameter `category` which we can capture using the `$_GET` superglobal and build the inner page on a category name basis.

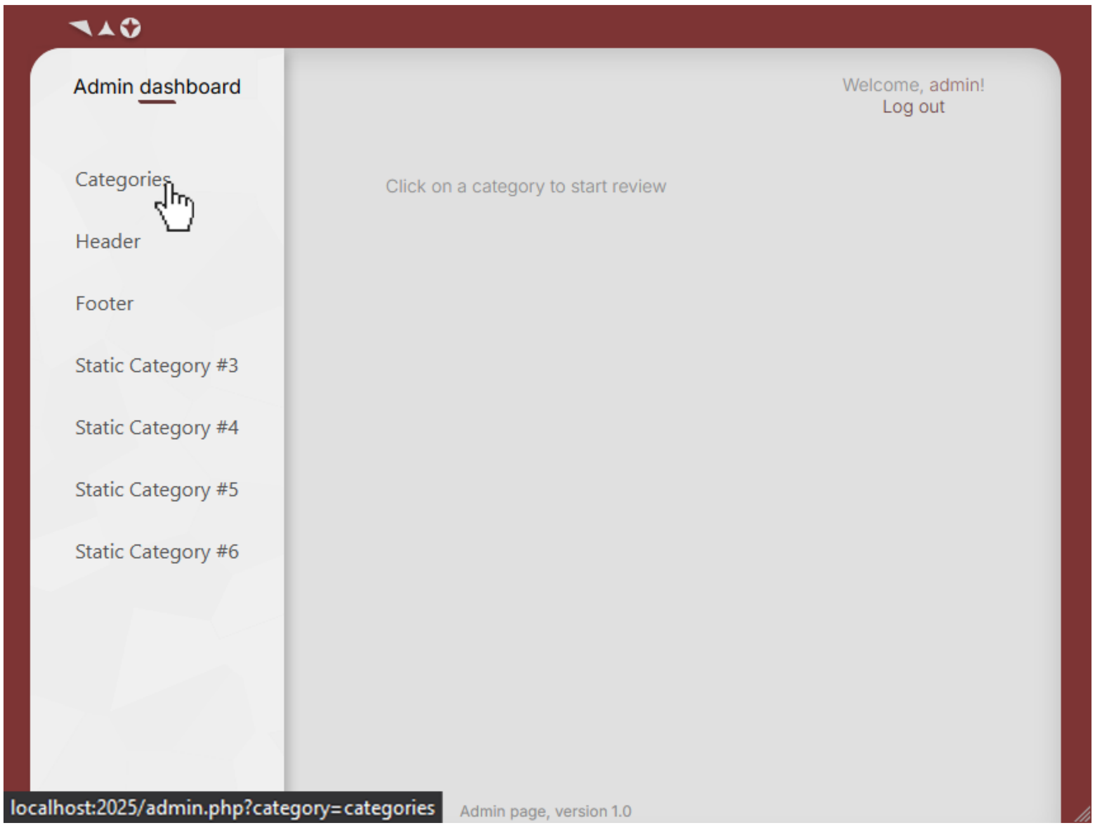


Figure 4.3.1: Default view, hovering over a category, in server, admin.php

In the above Figure (4.3.1) we can see what the framework will look like for the admin categories to be built on. Particularly, the section on the right can be overwritten to include the HTML that'll correspond to what we want each inner page to look like, while everything else remains the same. Assume the right section looks like the HTML in Listing 4.3.4, we can then build the inner page for each category in `$currentCategory` at the start of `admin.php` like shown in Listing 4.3.5.

```
...
switch ($buildHtml) {
    // Login screen / Failed login attempt
    case 1:
    ...
    // Logged-in framework
    case 2:
    ...
    $htmlBlock .= '<div class="right-inside">'. $currentCategory . '</div>
...
```

Listing 4.3.4: Right HTML section in logged in view, in server, admin.php

```
...
if (isset($_GET['category'])) {
    $currentCategoryName = $_GET['category'];
    switch ($currentCategoryName) {
        case "categories":
            $currentCategory = ... // HTML Build
        ...
        case $currentCategoryName === "header" ||
        $currentCategoryName === "footer":
            $currentCategory = ... // HTML Build
        ...
        default:
            $currentCategory = "Click on a category to start review";
...
```

Listing 4.3.5: Building admin category view, in server, admin.php

The category we'll start building first is "Categories". Like it was mentioned before, in subchapter 4.2, this category will provide the list of all available site categories. In addition to that, each item in the list will lead to yet another inner page which is where we'll be able to edit information such as the category title, excerpt, full text, etc.

To fetch all of the site categories associated with the admin category "Categories", we need to select them from the table `category_statuses`, and join both `admin_categories` and `site_categories` using each table's `id` field. This query can be seen in the database function `getSiteCategories()` shown in Listing 4.3.6.

```php
function getSiteCategories($c, $currentAdminCategory) {
    setDbSearchPathAdmin($c);
    $query = "SELECT site_categories.name, category_statuses.is_visible,
              category_statuses.currentorder, site_categories.id
              FROM category_statuses
              JOIN admin_categories ON category_statuses.admincategory_id =
              admin_categories.id
              JOIN site.site_categories ON
              category_statuses.sitecategory_id = site_categories.id
              WHERE admin_categories.name = '$currentAdminCategory'
              ORDER BY category_statuses.currentorder ASC";
    $statement = $c->prepare($query);
    $statement->execute();
    $result = $statement->fetchAll(PDO::FETCH_ASSOC);
    return $result;
}
```

Listing 4.3.6: Fetching site categories associated with "Categories", in server, db-functions.php

Generally, with `WHERE` we want to target IDs, or non-volatile objects, however in this particular example even if we changed the name of the admin category in the future it wouldn't be an issue with the query. Be confident in this being the case before going with this convention, however.

Before moving on to the next function, let's explain a concept about the query we just saw. Remember that `admin_categories` and `site_categories` each have a Primary Key (`id`) that is a Foreign Key in `category_statuses`. The Foreign Keys in and of themselves don't contain data about anything else but the ID itself which refers to the ID in the corresponding table. In order to get extra information, or more specifically columns, like `site_categories.name` or `admin_categories.name` which do not belong in the table `category_statuses`, we need to JOIN on those tables using the IDs that refer to the right records (see Datacamp, n.d., for more detail).

For queries that are complicated like in Listing 4.3.6, the SQL command in adminer helped to test and go through a lot of trial and error before getting it right, so executing queries through there is highly encouraged!

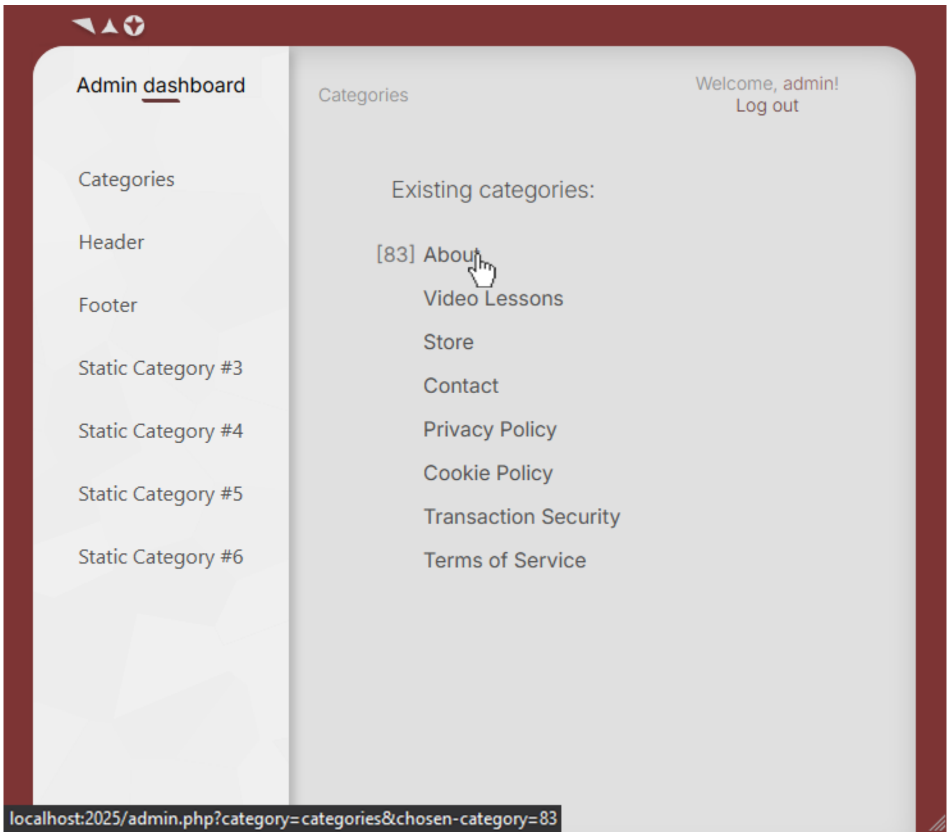
To read the values of `$result` we can use a for loop like before. To reference Listing 4.3.7, for each item we are building a link based on the ID of the site category, and showing the category's name as well as the ID when the item is hovered. Obtaining the ID in this way is handy if we ever need to reference it in the future when we are only interested in a single site category.

```
...
$dbResult = getSiteCategories($conn, $currentCategoryName);
if (empty($dbResult)) $currentCategory .= '<div class="no-available-
categories">No categories available. Add one!</div>';
else foreach ($dbResult as $item) {
       $currentCategory .= '<div class="category-item">
                             <a href="admin.php?category=' .
                             $currentCategoryName . '&chosen-category=' .
                             $item['id'] . '" class="category-item-
                             description">' . trim($item['name']) .
                             '<span class="category-item-hover-id">[' .
                             $item["id"].']</span></a></div>';
}
...
```

Listing 4.3.7: Using getsiteCategories(), in server, admin.php

In case there are no rows returned by the query in `$dbResult`, we can print a different message, notifying the user accordingly by checking if the array is empty.

In the next Figure (4.3.2) we can see what the view looks like for the "Categories" admin category. Make note of the link of the hovered site category:



localhost:2025/admin.php?category=categories&chosen-category=83

Figure 4.3.2: "Categories" view, hovering over a category, in server, admin.php

The next function we can discuss is `getSiteCategory()` from Listing 4.3.8. This function takes a site category ID as an argument, and returns information about that category, such as title, excerpt, full text, etc.

```
function getSiteCategory($c, $cId) {
    setDbSearchPathSite($c);
    $query = "SELECT id, name, titlefrontpage, titlecontentpage, excerpt,
              fulltext, permalink
              FROM site_categories
              WHERE id = $cId";
    $statement = $c->prepare($query);
    $statement->execute();
    $result = $statement->fetchAll(PDO::FETCH_ASSOC);
    return $result;
}
```

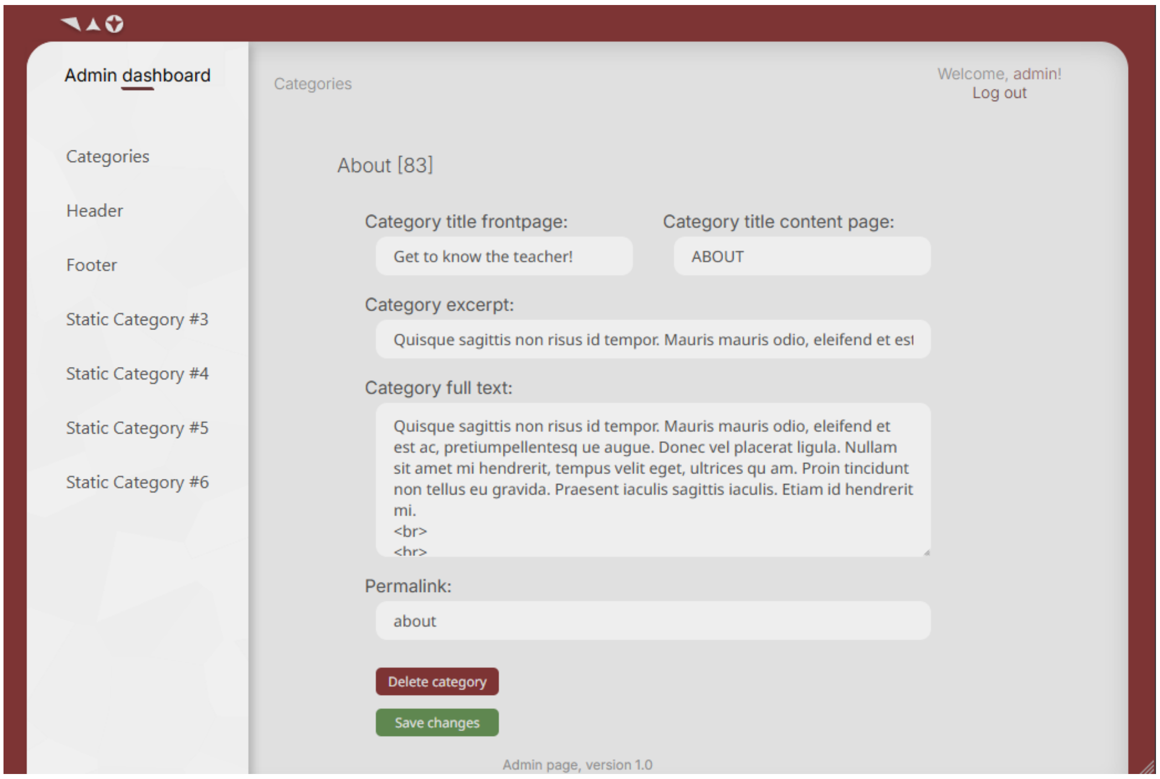
Listing 4.3.8: Fetching information about a single site category, in server, db-functions.php

In the admin page, we will use this information alongside a second function that instead sends information about a site category to the database (Listing 4.3.9).

```
function setSiteCategoryUpdate($c, $cId, $cTitleFp, $cTitleCp, $cExcerpt,
$cFullText, $cPermalink) {
    setDbSearchPathSite($c);
    $query = "UPDATE site_categories
              SET titlefrontpage = :cTitleFp, titlecontentpage = :cTitleCp,
              excerpt = :cExcerpt, fulltext = :cFullText, permalink =
              :cPermalink
              WHERE id = $cId";
    $statement = $c->prepare($query);
    $statement->bindParam(':cTitleFp', $cTitleFp);
    $statement->bindParam(':cTitleCp', $cTitleCp);
    $statement->bindParam(':cExcerpt', $cExcerpt);
    $statement->bindParam(':cFullText', $cFullText);
    $statement->bindParam(':cPermalink', $cPermalink);
    $statement->execute();
    return;
}
```

Listing 4.3.9: Updating a single site category's information, in server, db-functions.php

These two functions will be used together for the inner page that each site category item navigates to in “Categories”. The link we created earlier in Listing 4.3.7 plays a critical role here, as we can obtain the ID of the site category we navigated into by capturing it with `$_GET` on the inner page. Speaking of the inner page, the utility here will be changing the information of a site category, which is precisely what the query in Listing 4.3.9 does. However, we first fetch it with the query from Listing 4.3.8 to display it to the user.



About [83]

Category title frontpage:

Get to know the teacher!

Category title content page:

ABOUT

Category excerpt:

Quisque sagittis non risus id tempor. Mauris mauris odio, eleifend et est

Category full text:

Quisque sagittis non risus id tempor. Mauris mauris odio, eleifend et est ac, pretiumpellentesq ue augue. Donec vel placerat ligula. Nullam sit amet mi hendrerit, tempus velit eget, ultrices qu am. Proin tincidunt non tellus eu gravida. Praesent iaculis sagittis iaculis. Etiam id hendrerit mi.

Permalink:

about

Figure 4.3.3: Accessing a site category's information, in server, admin.php

In the above Figure (4.3.3) we can see what using the function from Listing 4.3.8 could look like. We have all of the information about the site category we clicked on, and from here we can execute the second function, from Listing 4.3.9, when we click on "Save changes". Since this particular function changes the state of the database (e.g. the site category data), this request will be done using a form with the POST method. This is consistent across the admin page – we want to allow users to freely bookmark frequently visited places on the page, without unknowingly duplicating data by visiting links that cause a change of state to either the page or the database (see Oded, 2013, for more detail). With that said, we will target `admin-functions.php`, and pass all of the information we fetched to use it in the update query, which will be in `db-functions.php`. Remember, in order to prevent CSRF with forms, we need to also include a token in this form as well.

As a side note before continuing, navigating to the inner page shown in Figure 4.3.3 can be made possible using an `if` statement in the switch statement case "categories" (seen in Listing 4.3.5) and checking whether or not `$_GET['chosen-category']` is set, and if it is to then build the inner page and break the switch statement before building the "Categories" admin category.

When we are sending the form containing the site category information to `admin-functions.php`, we can include a hidden input like before (in subchapter 4.1, Listing 4.1.2) to target the switch statement action with `flow-action`. The new case can be `category-update-single`, and the invocation of the query we set up in Listing 4.3.9 can be done with the code in Listing 4.3.10.

```
...
case "category-update-single":
    if ($_SESSION['session_admin_access'] == true) {
        $siteCategory          =     $_POST["category-to-update"];
        $currentAdminCategory = $_POST["current-admin-category"];
        $categoryTitleFp       =      $_POST["category-title-fp"];
        $categoryTitleCp       =      $_POST["category-title-cp"];
        $categoryExcerpt       =       $_POST["category-excerpt"];
        $categoryFullText      =     $_POST["category-full-text"];
        $categoryPermalink     =     $_POST["category-permalink"];
        setSiteCategoryUpdate($conn, $siteCategory, $categoryTitleFp,
        $categoryTitleCp, $categoryExcerpt, $categoryFullText,
        $categoryPermalink);
        header('Location: admin.php?category=' . $currentAdminCategory .
        '&chosen-category=' . $siteCategory . '&changes-saved=true');
        exit;
    } else header("Location: admin.php");
...
```

Listing 4.3.10: Directing site category data to setSiteCategoryUpdate(), in server, admin-functions.php

One thing to mention here about Listing 4.3.10, is that after invoking the database function `setSiteCategoryUpdate()`, we redirect the flow back to the same view we had before clicking "Save changes", shown in Figure 4.3.3, with the `header()` function.

The next function is deleting the site category, which your eye may have already caught in Figure 4.3.3. The idea is simple: we delete the row from `site_categories`. The row being the current site category we are viewing. Again, while this is a database function and lives in `db-functions.php`, since it causes a change in data we send a POST request to `admin-functions.php` first. The function that deletes the record can be seen in Listing 4.3.11.

```
function setSiteCategoryDelete($c, $cId) {
    setDbSearchPathSite($c);
    $query = "DELETE
              FROM site_categories
              WHERE id = $cId";
    $statement = $c->prepare($query);
    $statement->execute();
    return;
}
```
Listing 4.3.11: Deleting a record from site_categories, in server, db-functions.php

The code that invokes this function from `admin-functions.php`, can be seen in Listing 4.3.12.

```
...
case "category-delete-single":
    if ($_SESSION['session_admin_access'] == true) {
        $siteCategory          =             $_POST["category-for-deletion"];
        $currentAdminCategory = $_POST["current-admin-category-secondary"];
        setSiteCategoryDelete($conn, $siteCategory);
        header('Location: admin.php?category=' . $currentAdminCategory);
        exit;
    } else header("Location: admin.php");
...
```
Listing 4.3.12: Delete record directive, in server, admin-functions.php

Unfortunately if testing these kind of SQL queries in SQL command in adminer, the records will actually be removed, so be careful with testing.

This is the full utility the “Categories” admin category provides so far. A potential change in the future could be allowing the user to rename the site category from the level 3 inner page.

Moving on to the “Header” and “Footer” admin categories. These two can be treated as the same case in admin.php. Since `getSiteCategories()` from Listing 4.3.6 takes an admin category as an argument, the list returned for each one will have different attributes. A couple things we didn’t utilize the first time around we used that particular function, were the columns `is_visible` and `currentorder` from the table `category_statuses`. Both of these come to the forefront of what “Header” and “Footer” provide as utility. The idea with either one, is to control which site categories will appear in the header and footer sections on the website, as well as in which order. This is possible because each admin category retains its own status of each site category in the table.

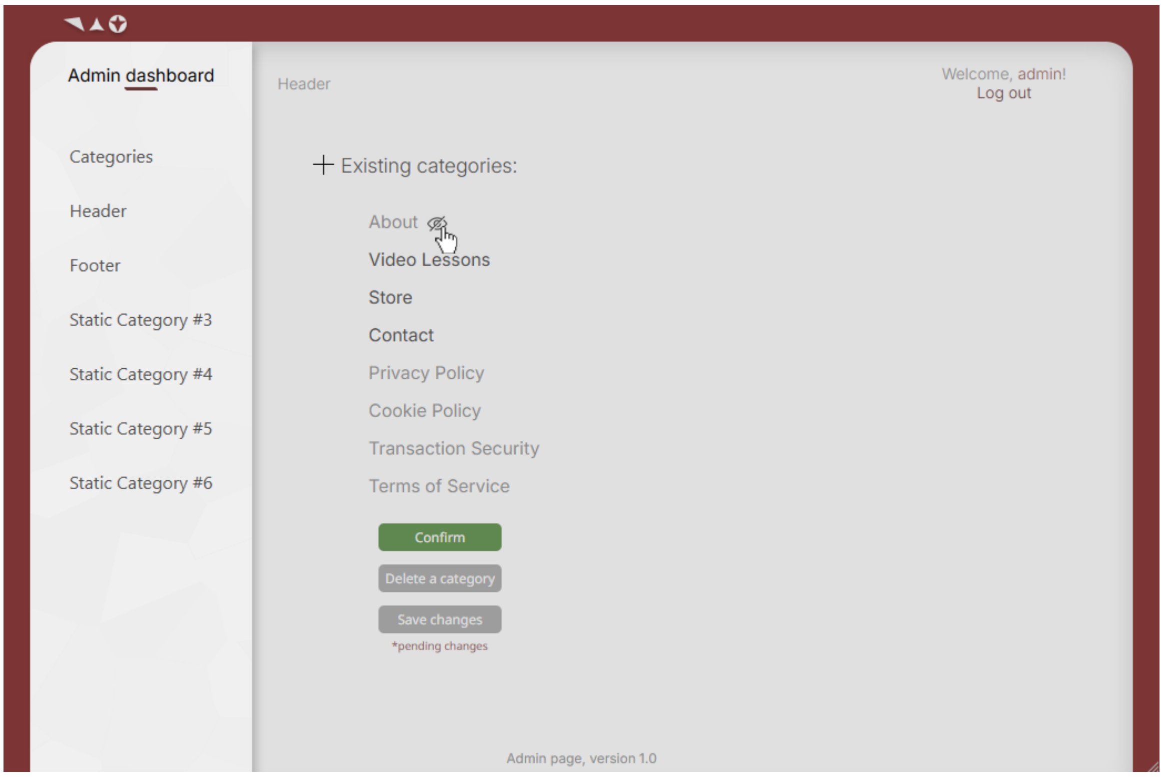


Figure 4.3.4: "Header" admin category view, hovering over visibility setting, in server, admin.php

In the above Figure (4.3.4) we can see the column `is_visible` being utilized to set the visibility of each site category in this particular admin category, which happened to be “Header”. On this view, the user has a few options:

- Set category visibility
- Change category order
- Delete categories
- Create new category

All of the above options can be done one by one, or if the user wishes, in bulk, with each save. Every action done simply marks a category item appropriately, and when “Save changes” is clicked, all of the marked items are sent with a form using the POST method to `admin-functions.php`. This particular view was quite tricky to finalize simply because of all the different variables I had to account for in the bulk option submission. For example, if the user creates a new category but for some reason chooses to mark it for deletion, then this item should not be included in the form at all. Another example is if the user changes the visibility of a category to hidden but also deletes it, the database procedure should only involve dropping that record since the visibility does not matter if the category does not even exist. Many use cases like that had to be considered during this phase, and I'm sure I even missed a couple.

The function that enables all of this data to be written into the database is a little over 100 lines of code, so suffice to say that I won't be including it here in full, but we can partly see it in Listing 4.3.13.

```php
function setSiteCategories($c, $cId ,$cName, $cIsRemoved, $cIsVisible,
$cOrder, $cIsNew, $currentAdminCategory) {
    if ($cIsRemoved == "true") {
        setDbSearchPathSite($c);
        $query = "DELETE
                  FROM site_categories
                  WHERE id = :cId";
        ...
    } else if ($cIsRemoved == "false") {
        setDbSearchPathAdmin($c);
        $query = "UPDATE category_statuses
                  SET is_visible = $cIsVisible, currentorder = $cOrder
                  FROM site.site_categories, admin_categories
                  WHERE category_statuses.sitecategory_id =
                  site_categories.id
                  AND category_statuses.admincategory_id =
                  admin_categories.id
                  AND site_categories.id = :cId
                  AND admin_categories.name = '$currentAdminCategory'";
        ...
    }
    if ($cIsNew == "true") {
        setDbSearchPathSite($c);
        $buildPermalink =                                trim($cName);
        $buildPermalink =  str_replace(" ", "-", $buildPermalink);
        $buildPermalink = str_replace("--", "-", $buildPermalink);
        $buildPermalink = mb_strtolower($buildPermalink, 'UTF-8');
        $cPermalink     =          greekToAscii($buildPermalink);
        $query = "INSERT INTO site_categories (name, permalink)
                  VALUES (:cName, :cPermalink)";
        ...
```

Listing 4.3.13: Snippet of multiple queries for "Header" and "Footer" admin categories, in server, db-functions.php

Specifically, we can see the query for deleting a category, updating the visibility and order, and finally adding a new category, which also sets the permalink for the user automatically in a combination of using the name of the category and removing unnecessary whitespaces before and after the word(s) and replacing the rest with dashes.

The function, as mentioned already, is invoked by `admin-functions.php` with the code found in Listing 4.3.14.

```
...
case "categories-update":
    if ($_SESSION['session_admin_access'] == true) {
        $categoryCount          = $_POST["category-count"];
        $currentAdminCategory =     $_POST["return-url"];
        for($i = 0; $i < $categoryCount; $i++) {
            $postPrefix = 'category-item-posting-' . $i . '-';
            if ($_POST[$postPrefix . 'name'] != null) {
                $categoryId         =         $_POST[$postPrefix . 'id'];
                $categoryName       =       $_POST[$postPrefix . 'name'];
                $categoryIsRemoved =    $_POST[$postPrefix . 'removed'];
                $categoryIsVisible = $_POST[$postPrefix . 'visibility'];
                $categoryOrder      =      $_POST[$postPrefix . 'order'];
                isset($_POST[$postPrefix . 'is-new']) ? $categoryIsNew =
                true : $categoryIsNew = false;
                setSiteCategories($conn, $categoryId, $categoryName,
                $categoryIsRemoved, $categoryIsVisible, $categoryOrder,
                $categoryIsNew, $currentAdminCategory);
            }
        }
        header('Location: admin.php?category=' . $currentAdminCategory .
       '&changes-saved=true');
        exit;
    } else header("Location: admin.php");
...
```

Listing 4.3.14: Form handling of multiple category update, in server, admin-functions.php

Each site category item sent over for processing with the form is iterated on with a `for` loop. Other than making sure the `for` loop runs in bounds with `$categoryCount`, and marking new categories with `$categoryIsNew = true`, we process all of the information about the `$i`th category then pass it to `setSiteCategories()`. It's unfortunate that most of the hard work around this has to do with how the category item marking is done with Javascript in `admin.php`, but that's all outside of the scope for this chapter and won't be discussed.

With that, all of the functionalities the admin page has to offer are concluded. From here there are a ton of things we can do, and that's why it was important to keep scalability in mind and was why in turn we saw a lot of switch statement usage.

5 "Cleaning up"

We have almost reached the end of the document, but there are still some important things left to discuss. Currently, all of the PHP files we have created in the container are all sitting in `/var/www/html/` and are easily accessible by a client right through the browser. This is not a problem for the pages we want rendered, like `index.php`, but is unnecessarily exposing backend files and potentially increasing the attack surface of our website. Additionally, in order to access the pages we do want rendered, it's required that we access their corresponding files manually by typing out the file name with its extension (.php), when traditionally, at least as the user, we are used to more clean URLs in navigation. Lastly, we are creating more and more files in the Trusted Entity's sink every time a new client accesses the website without disposing them appropriately. In the coming pages we will address each topic separately, in concise, focused subchapters.

5.1 Backend file protection

Opening up with the first issue, putting order in the webroot (or document root – the root folder the web server looks for index) is an important part for ease of maintenance but also ensures that only the files absolutely necessary become public-facing (i.e. accessible by clients). A great example for this is how the Laravel (n.d.) framework structures file directories to separate core code, configuration files, tests, resources, etc. For our purposes, separating code that should be immediately accessible by clients, like `index.php`, and `admin.php` from backend specific files like `db-functions.php`, or backend operations, like the API chain we made in chapter 3, will be sufficient.

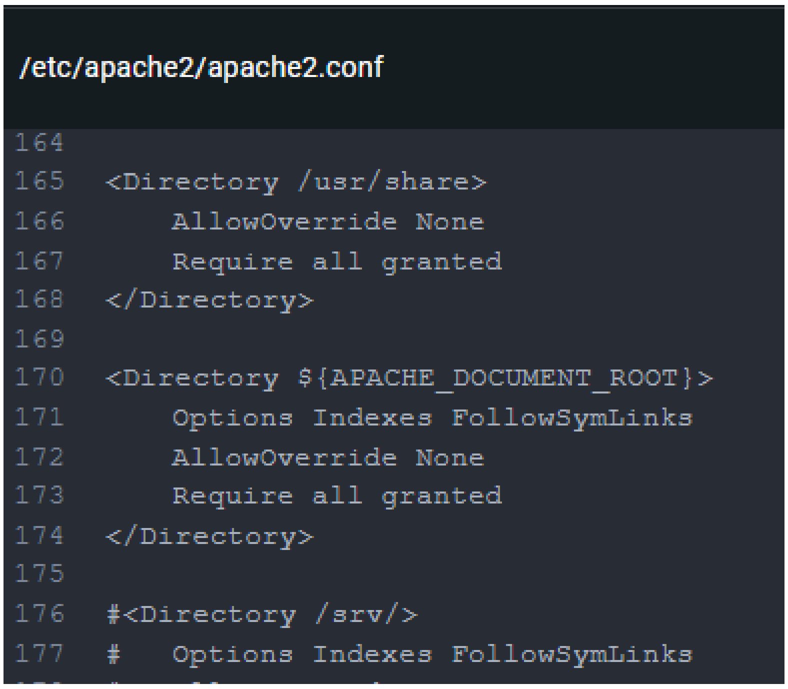
The way we'll do this involves a little more work than simply having the different files in two separate folders in `/var/www/html/`. If we did this, any one of those files would still be accessible by clients in the browser, by typing out the path to the file, for example `localhost:2025/folder/backend-file.php`. Instead, we'll change the webroot apache assigns by default, which is `/var/www/html/` to `/var/www/html/public`, then move all of the sensitive files in a different folder, like `/var/www/html/private`. This way, we have moved those sensitive files outside of the webroot's context, and have prevented direct access to them through the browser.

So, how do we change the webroot? If we were to do it manually, we'd have to change multiple Apache configuration files, including `apache2.conf`. To do this while keeping the container as ephemeral as possible, we can change all mentions of the directory in all of the target files, inside the Dockerfile.

```
# syntax=docker/dockerfile:1
# Server
FROM php:8.2.10-apache
# Change Document Root
ENV APACHE_DOCUMENT_ROOT /var/www/html/public
RUN sed -ri -e 's!/var/www/html!${APACHE_DOCUMENT_ROOT}!g'
/etc/apache2/sites-available/*.conf
RUN sed -ri -e 's!/var/www/!${APACHE_DOCUMENT_ROOT}!g'
/etc/apache2/apache2.conf /etc/apache2/conf-available/*.conf
...
```

Listing 5.1.1: Changing the webroot of the server container

The code in Listing 5.1.1, which can be found on the official PHP image's Docker Hub page, substitutes all mentions of the default directory with the new webroot directory specified in `APACHE_DOCUMENT_ROOT`. The changes are reflected in Figure 5.1.1:


/etc/apache2/apache2.conf

```

<Directory /usr/share>
    AllowOverride None
    Require all granted
</Directory>

<Directory ${APACHE_DOCUMENT_ROOT}>
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>

#<Directory /srv/>
#   Options Indexes FollowSymLinks
```

Figure 5.1.1: Webroot directory after Dockerfile change

Now any files that reside in `/var/www/html/public` can be accessed by clients, while anything that is in `/var/www/html/private` remains out of direct reach. One way we can test this is by typing out the URL `localhost:2025/public/../private/backend-file.php`, and seeing that the browser dismisses the attempt to go backwards in the path.

For public-facing PHP files that need access to files outside of webroot, this is possible with relative path traversal, like `require __DIR__.'/../private/db-functions.php'`.

Of course, separating files in this way does not completely secure our server, however it greatly reduces its attack surface, allowing us to focus on the security of the few files that consist of that surface.

5.2 PHP file routing

Next, we will be alleviating the need for file paths and extensions in URLs, by defining the paths as routes in a centralized space. This change will allow a client to navigate to pages without needing to include the file extension, which doesn't immediately add many benefits other than URLs having a more human readable format, and apparently does not improve Search Engine Optimization (SEO; see Southern, 2018, for more detail). Personally, I am of the belief that uniformity of URLs on the web help with user experience, and seeing that most websites have URLs that follow this trend, then I think that's a good enough reason to implement this change.

The way we'll be achieving this with Apache, is by expanding on the configuration file that was shown in Figure 5.1.1 from the last subchapter. By enclosing a set of directives within the named directory, we can control how the server behaves for that particular directory. In specific, we will use the RewriteRule directive to rewrite all requests in such a way that we can utilize the `$_GET` superglobal to capture those requests and route them to the files manually.

```
...
# Routing rule
RUN sed -i '/<Directory ${APACHE_DOCUMENT_ROOT}>/,/<\/Directory>/d'
/etc/apache2/apache2.conf
RUN echo '\
\n<Directory ${APACHE_DOCUMENT_ROOT}>\n\
  Options FollowSymLinks\n\
  AllowOverride None\n\
  Require all granted\n\
\n\
  RewriteEngine On\n\
  RewriteRule ^([a-zA-Z0-9-]+)$ index.php?route=$1 [QSA,L]\n\
</Directory>\n' >> /etc/apache2/apache2.conf
# Enable mod_rewrite module (Rewrite Enginge)
RUN a2enmod rewrite && \
    service apache2 restart
...
```

Listing 5.2.1: Expanding on the server configuration file

After we substitute the old webroot path with the new one as shown in Listing 5.1.1, we delete the entire block that corresponds to the webroot directive for `/var/www/html/public`, and re-introduce it with the new directive in place as shown in Listing 5.2.1. The directive itself is a regular expression that matches URL requests with a pattern containing any alphanumeric characters, including hyphens, and rewrites it in the format of `index.php?route=$1`, where `$1` is the captured group in the request.

Simply put, a request to our server like `localhost:2025/admin` would be rewritten as `localhost:2025/index.php?route=admin`. Now, in `index.php` we can capture `route` and decide which file the server should response with.

```php
<?php
$destination = $_GET["route"] ?? '';
    switch ($destination) {
        case "":
            include "frontpage.php";
            break;
        case "admin":
            include "admin.php";
            break;
        default:
            include "404.php";
    }
```

Listing 5.2.2: Routing examples, in server, index.php

In Listing 5.2.2, we can see the routing in action, in our `index.php` file. In any case the captured `route` does not correspond to a defined file, the `404.php` file will be returned. If the captured group is empty (e.g. `localhost:2025/`), `index.php` will respond with `frontpage.php`, which is the old `index.php` file.

With this framework it's easy to scale the size of the website, whether that means needing to add or remove pages, as well as utilize the permalink of each category that was shown in the last chapter.

It is to be noted that while this implementation can also be done using `.htaccess` files, it is recommended to stick to server configuration files if those are available to us. The performance hit of the server needing to look in multiple directories for `.htaccess` files, as well as those files needing to be fetched with every request, make changes done in the main configuration file a lot more sensible if we have access to it (see Apache, n.d., for more detail).

5.3 Cron & Garbage Collection

We may have reached the last task this Thesis will be covering, but this is by no means an exhaustive list of the steps required in our work to build a robust server. If anything, this article has hopefully inspired to build upon these fundamentals, or use any number of them to produce a transformative piece of work that meets the proper requirements or needs of the reader's project.

To get back on topic, let's start by reminding of how the trustedentity container we built in chapter 2, particularly for the purpose of introducing Secret Zero back to the server container using a RabbitMQ RPC described in subchapter 2.4, is currently creating files in a common volume called "tmpfs-shared" which bear the wrapped Secret Zero. For what it's worth, any client requesting access to our website will have an active session, therefore trustedentity will keep writing into the file that corresponds to the specific custom session ID belonging to that client, but it won't get rid of the file when the session becomes inactive or is otherwise lost.

Since we are using a tmpfs volume, and as mentioned before, it will lose any data in it, including those session based files, once the containers that it depends on are both shut down. However, unless we are counting on our server and trustedentity dying on purpose, these particular session files will keep on building up in tmpfs.

One way to combat this, is by scheduling a garbage collection task using Cron. This binary will run in the background in trustedentity, while the message listener continues running in the foreground. For those that have been paying attention, they'll remember that in chapter 1 we discussed that it's a best practice to build containers in Docker while thinking of them as single service objects. Therefore, the idea of running two tasks in trustedentity betrays that principle, but honestly, only barely. The decision to run Cron as the secondary, background service in trustedentity, only adds the risk of us not being notified in the case that Cron dies, which would cause us to miss out on the garbage collection task it offers. This risk is minimal, but ultimately it's up to the developer to weigh the pros and cons of this implementation.

With that said, let's look at how we can introduce Cron and the garbage collection task to our trustedentity container. In Listing 5.3.1, we install the binary, then write the task for Cron to find, in `/etc/cron.d/`.

```
...
# Garbage Collection (cron)
RUN apt-get update && apt-get -y install cron
RUN echo "0 * * * * sh /cron/garbage-collect.sh > /proc/1/fd/1
2>/proc/1/fd/2" > /etc/cron.d/garbage-collect
RUN chmod 0640 /etc/cron.d/garbage-collect
RUN crontab -u www-data /etc/cron.d/garbage-collect \
    && chmod u+s /usr/sbin/cron
...
ENTRYPOINT ["/usr/bin/entrypoint.sh"]
```

Listing 5.3.1: Introducing Cron and its main task in trustedentity's Dockerfile

The actual task is to run a script (Listing 5.3.2) that does the garbage collection, once every 1 hour. Something important about the task is that when the script is executed, its `stdout` and `stderr` outputs are routed to Docker's main process logs (see HugoShaka, 2017, for more detail). We can use this to log any desired events after running the garbage collection script.

In the Dockerfile (Listing 5.3.1) we associate the Cron task with the user `www-data`, which is the same user that we want to run as in the container, but elevate Cron's permissions with `chmod u+s /usr/sbin/cron` which effectively sets the `setuid` for the Cron binary. What this means, is that essentially Cron is running with root permissions but the user we run as in the container is `www-data`, and it's the best compromise we can resort to if we want to use Cron but run as a non-root user in the container. We discussed wanting to avoid running as root user in our containers previously, and while this is not exactly the same, we could potentially be introducing vulnerabilities by doing this, so use Cron with discretion.

```
#!/bin/bash
start_time=$(date +%s%N)
shared_files="/pipe"
total_count=$(find "$shared_files" -type f -mmin +10 | wc -l)
find "$shared_files" -type f -mmin +10 -exec rm -f {} +
end_time=$(date +%s%N)
elapsed_ns=$((end_time - start_time))
elapsed_ms=$((elapsed_ns / 1000000))

echo "Garbage collected $total_count files from $local_files and
$shared_files. Time taken: $elapsed_ms ms"
```

Listing 5.3.2: Garbage collection script

Looking at the script itself from Listing 5.3.2, along with some statistics like files affected and time elapsed, we search in the common volume `/pipe/` for any files that have not been modified in longer than 10 minutes. In the same command, we force remove those files with `rm -f`.

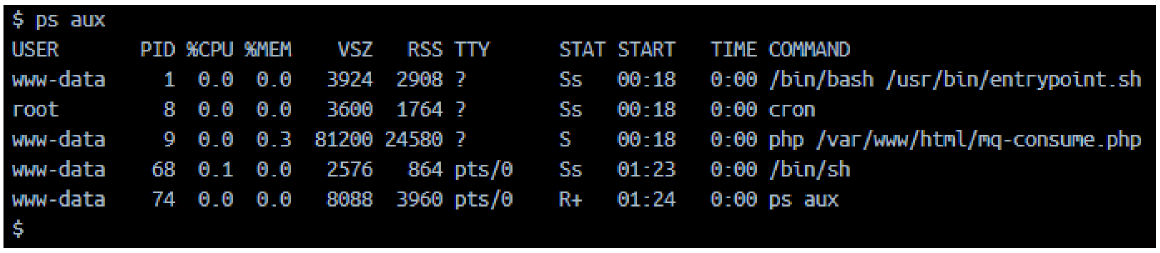
Since the Cron task's output is redirected to Docker's logs, the `echo` command in the script will print that particular message for us to find in those logs.

That settles configuring the task, and scheduling it to run on an interval. To set Cron as the background task and the message listener to run in the foreground, we'll need to update the old entrypoint (which it currently is `["php", "/var/www/html/mq-consume.php"]`) to reflect that. Instead of setting the entrypoint in the compose file, we can do so in the Dockerfile, after switching to user `www-data`. Specifically, we can define the entrypoint configuration in a file (Listing 5.3.3), copy it in the container, and then assign the path of that configuration as the entrypoint file (Listing 5.3.4).

```bash
#!/bin/bash
# Cron jobs
cron
echo "Cron Service Started.."
php /var/www/html/mq-consume.php
```
Listing 5.3.3: Trustedentity's entrypoint file (entrypoint.sh)

```dockerfile
...
# Build + Ownership, includes /usr/bin/entrypoint.sh
WORKDIR /
COPY --chown=www-data:www-data . .
# Perms
RUN chmod 0500     /cron/garbage-collect.sh
RUN chmod 0500       /usr/bin/entrypoint.sh
...
# Switch User
USER www-data
# Entrypoint file
ENTRYPOINT ["/usr/bin/entrypoint.sh"]
```
Listing 5.3.4: Setting up trustedentity's entrypoint, in its Dockerfile

And that's it. As long as `mq-consume.php` keeps listening for messages on the queue, the container will keep running, and Cron should be working on the background.



```
$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
www-data     1  0.0  0.0   3924  2908 ?        Ss   00:18   0:00 /bin/bash /usr/bin/entrypoint.sh
root         8  0.0  0.0   3600  1764 ?        Ss   00:18   0:00 cron
www-data     9  0.0  0.3  81200 24580 ?        S    00:18   0:00 php /var/www/html/mq-consume.php
www-data    68  0.1  0.0   2576   864 pts/0    Ss   01:23   0:00 /bin/sh
www-data    74  0.0  0.0   8088  3960 pts/0    R+   01:24   0:00 ps aux
$
```

Figure 5.3.1: Fetching processes running in trustedentity

In Figure 5.3.1, it's shown that as long as we can see Cron and `mq-consume.php` on the list `ps aux` returns, we know that Cron had sufficient permissions to start, and `mq-consume.php` is running in the foreground. To test that Cron is indeed a background task and that `mq-consume.php` is running in the foreground, we can run the command `kill -9 PID`, where PID is the process ID assigned to each task. We should see that killing Cron doesn't affect the container, but killing `mq-consume.php` shuts it down (or restarts it, depending on the `restart` directive in compose.yaml).

6 References

A. Cheshirov. (2012, November 3). *Note: str_shuffle() internally uses rand(), which is unsuitable for cryptography purposes (e.g. generating random passwords). You want a secure random* [Comment on the online forum post *PHP random string generator*]. StackOverflow. https://stackoverflow.com/a/13212994

Apache. (n.d.). *Apache HTTP Server Tutorial: .htaccess files*. Retrieved January 25, 2025, from https://httpd.apache.org/docs/2.4/howto/htaccess.html

Datacamp. (n.d.). *JOIN tables linked by a foreign key*. Retrieved January 23, 2025, from https://campus.datacamp.com/courses/introduction-to-relational-databases-in-sql/glue-together-tables-with-foreign-keys?ex=4

Docker. (n.d.-a). *Building best practices*. Retrieved January 6, 2025, from https://docs.docker.com/build/building/best-practices

Docker. (n.d.-b). *Docker Desktop WSL 2 backend on Windows*. Retrieved January 4, 2025, from https://docs.docker.com/desktop/features/wsl

GeeksForGeeks. (2024, June 21). *Docker Tips: about /var/run/docker.sock*. https://www.geeksforgeeks.org/docker-tips-about-varrundockersock

Guzzlephp. (n.d.-a). *Request Options.* Retrieved January 15, 2025, from https://docs.guzzlephp.org/en/latest/request-options.html#json

HashiCorp. (n.d.-a). *Response wrapping.* Retrieved January 14, 2025, from https://developer.hashicorp.com/vault/docs/concepts/response-wrapping

HashiCorp. (n.d.-b). *Seal/Unseal.* Retrieved January 13, 2025, from https://developer.hashicorp.com/vault/docs/concepts/seal

HashiCorp. (n.d.-c). *Vault configuration parameters*. Retrieved January 11, 2025, from https://developer.hashicorp.com/vault/docs/configuration#disable_mlock

HugoShaka. (2017, September 14). *The accepted answer may be dangerous in production environment. In docker you should only execute one process per container* [Comment on the online forum post *How to run a cron job inside a docker container?*]. StackOverflow. https://stackoverflow.com/a/46220104

Laravel. (n.d.). *Directory Structure*. Retrieved January 24, 2025, from https://laravel.com/docs/11.x/structure

Oded. (2013, March 1). *This is not advice. A GET is defined in this way in the HTTP protocol. It is supposed to be* [Comment on the online forum post *Why shouldn't a GET request change data on the server?*]. StackOverflow. https://softwareengineering.stackexchange.com/a/188861

Oneupme. (2024, September 16). *I will describe that specific letter you sent. Your mailman picks it up from your mailbox and brings it back* [Comment on the online forum post *ELI5: How does the USPS deliver letters in the mail?*]. Reddit. https://www.reddit.com/r/explainlikeimfive/comments/1fifjql/eli5_how_does_the_usps_deliver_letters_in_the_mail/lnh6tr3

Owasp. (n.d.-a). *Cross-Site Request Forgery Prevention Cheat Sheet.* Retrieved January 20, 2025, from https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

Php. (n.d.-a). *hash_equals*. Retrieved January 22, 2025, from https://www.php.net/manual/en/function.hash-equals.php

Php. (n.d.-b). *random_bytes*. Retrieved January 22, 2025, from https://www.php.net/manual/en/function.random-bytes.php

PostgreSQL. (n.d.-a). *Schemas*. Retrieved January 19, 2025, from https://www.postgresql.org/docs/current/ddl-schemas.html#DDL-SCHEMAS-PRIV

RabbitMQ. (n.d.-a). *Queues*. Retrieved January 17, 2025, from https://www.rabbitmq.com/docs/queues#durability

RabbitMQ. (n.d.-b). *RabbitMQ tutorial - Publish/Subscribe*. Retrieved January 17, 2025, from https://www.rabbitmq.com/tutorials/tutorial-three-dotnet#exchanges

Shiflett C. (2004, February 16). *Session Fixation*. https://shiflett.org/articles/session-fixation

Shital Shah. (2016, January 13). *It might be helpful to understand how virtualization and containers work at a low level. That will clear up lot* [Comment on the online forum post *How is Docker different from a virtual machine?*]. StackOverflow. https://stackoverflow.com/a/34757096

Southern, M. G. (2018, March 31). *Google's John Mueller Says File Extensions in URLs Do Not Matter*. https://www.searchenginejournal.com/googles-john-mueller-says-parameters-urls-not-matter/246675

W3Schools. (n.d.-a). *HTML <pre> tag*. Retrieved January 18, 2025, from https://www.w3schools.com/tags/tag_pre.asp

W3Schools. (n.d.-b). *PHP MySQL Prepared Statements*. Retrieved January 18, 2025, from https://www.w3schools.com/php/php_mysql_prepared_statements.asp