Microservices and Web Application Architecture

Cybersecurity

Web Development Day 2

Class Objectives

By the end of today's class, we'll be able to:

Understand how microservices and architecture work to deliver more robust, reliable, and repeatable infrastructure as code.

Define the different services within a LEMP stack.

Deploy a Docker Compose container set and test the deployment functionality.

Describe how relational databases store and retrieve data.

Application Structure

Introduction to Microservices

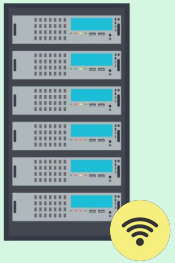
Today, we'll set up and secure microservices.

First, we need to understand the following:

Components of a Typical Web App

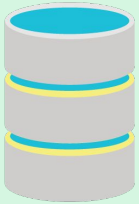
A developer, Andrew, often uses a **browser to manage a public-facing employee directory application** to retrieve and update different employee lists from his company.

- The employee directory application is set up so that Andrew can add, remove, and view employees.
- The employee directory application needs to access the following components:



Front-end server
To display webpages and style them in readable formats. Also responsible for receiving and responding to HTTP requests.

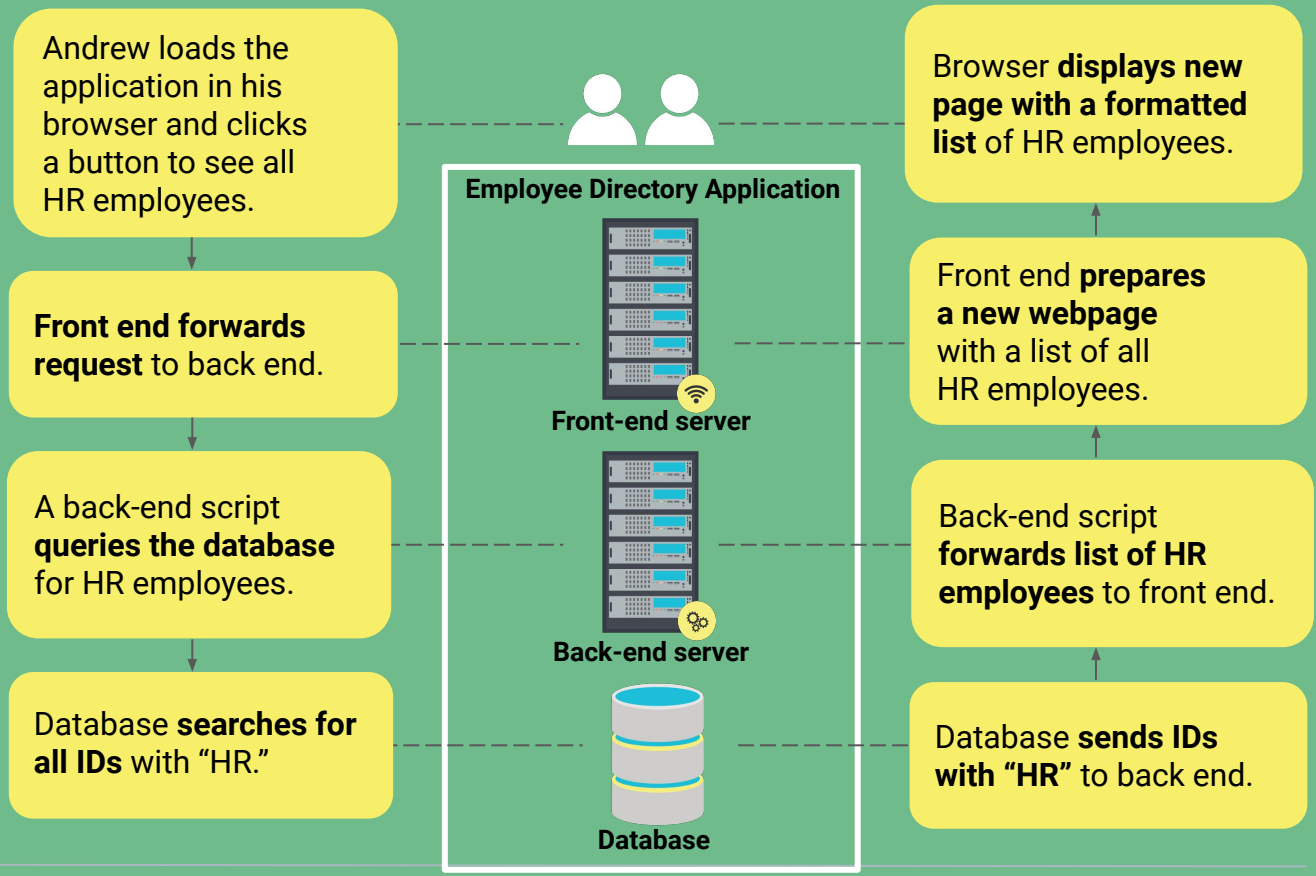
Back-end server
Executes business logic and writes or reads corresponding data to and from a database.



Database
Stores information about employees, such as their employee IDs and names. Info can be critical and sensitive.

Information Flows Between Application Components

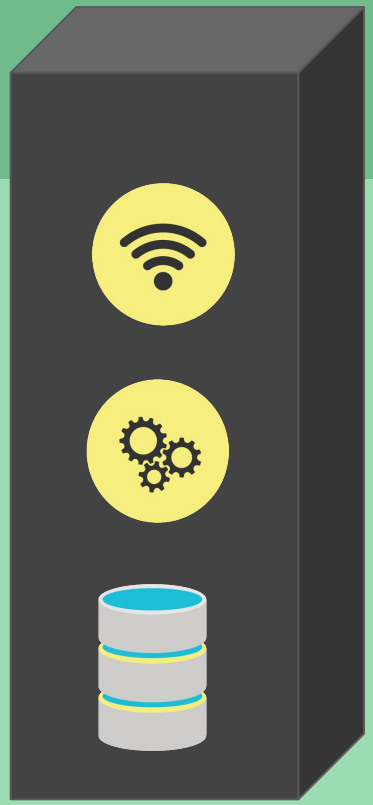
Now that we know the main components of a web application, we want to know what happens behind the scenes when a user interacts with a web application, or more specifically, makes an HTTP request to the server.



Monolith to Microservices

Understanding application components is key to understanding the modern architectural paradigm of microservices.

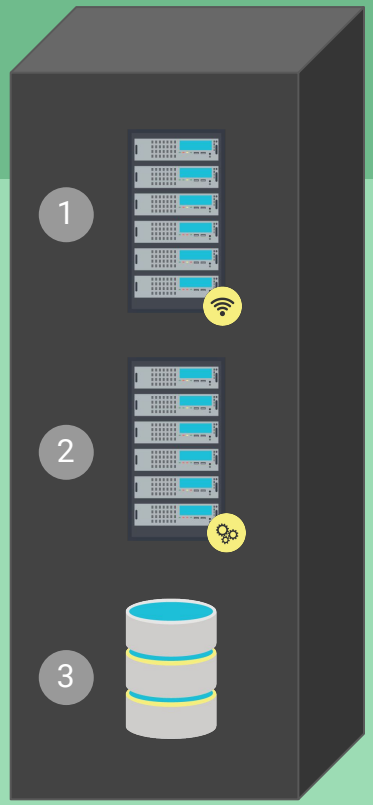
Let’s first explore the original architecture type, the monolith, and the issues that led to the adoption of microservices.



Monolith

A monolith is a **single machine** that **hosts all the components required to serve a website or application.**

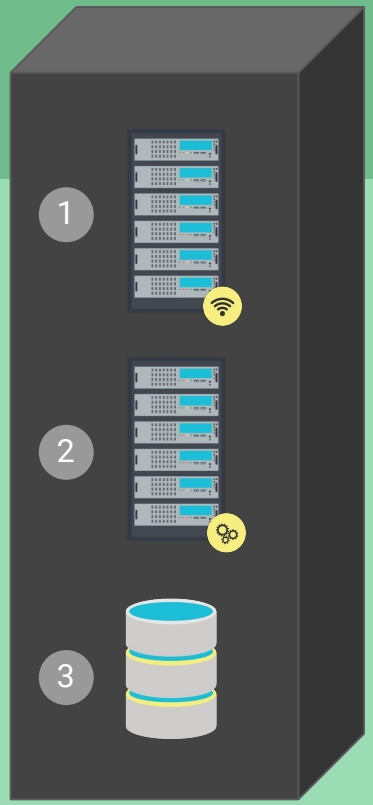
In other words, a monolith is a machine that has a front-end server, back-end server, and a database.



Monolith

Suppose Amazon is using a monolithic server. It would contain:

1. **Front-end HTML server**: GUI for customers to look at while shopping.
2. **Back-end application server**: Back-end server showing the inventory and stock. Interacts directly with the database.
3. **MySQL database**: Database of customers, and their information and purchases.



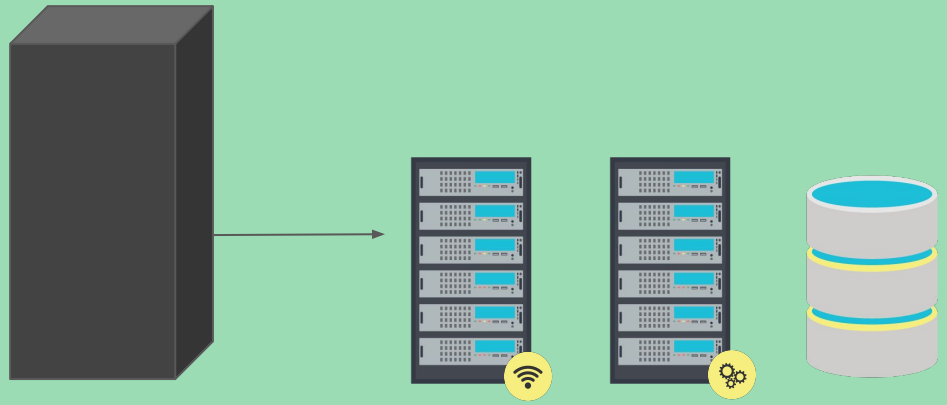
Monolith

The components of a single machine are highly dependent on each other. If any component malfunctions, the entire application will malfunction.

- **Hackers, environmental issues, and human errors** can threaten the entire machine.
- **Updating a single component means taking down all components**, resulting in downtime and lack of availability.

Problems with Monoliths

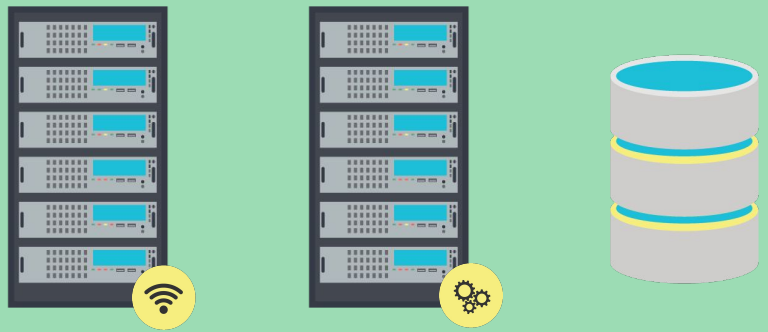
It’s now expected that companies will ensure availability by maintaining almost 100% uptime. The risk associated with a interdependent monolithic system presented problems.



- Separating an application's components into their own machines (**microservices**) means the sysadmin can update the front end, reboot it, and only have to verify the front-end components—not the back-end components and database.
- If a component is compromised by a hacker, the potential damage is restricted to only that component.

Microservices

Each independent machine is a component that executes one primary function or **service.**



We already implemented microservices when we covered cloud security:

- A **jump box** to connect to a private Ansible server.
- An **Ansible server** to configure DVWA servers.
- **DVWA servers** to provide the DVWA service.
- A **Docker image** to provide WordPress.

Benefits of Microservices

Scalability and resilience	Rapid response	Isolated improvement	Isolated security
Replication of components lets you serve more clients and provides identical backup components if one fails.	Microservice components are smaller, so they can be replaced and updated quickly.	Microservices serve one primary function, so they can be developed with the goal of optimizing their functionality.	One compromised component does not equal a compromised application.

APIs

APIs

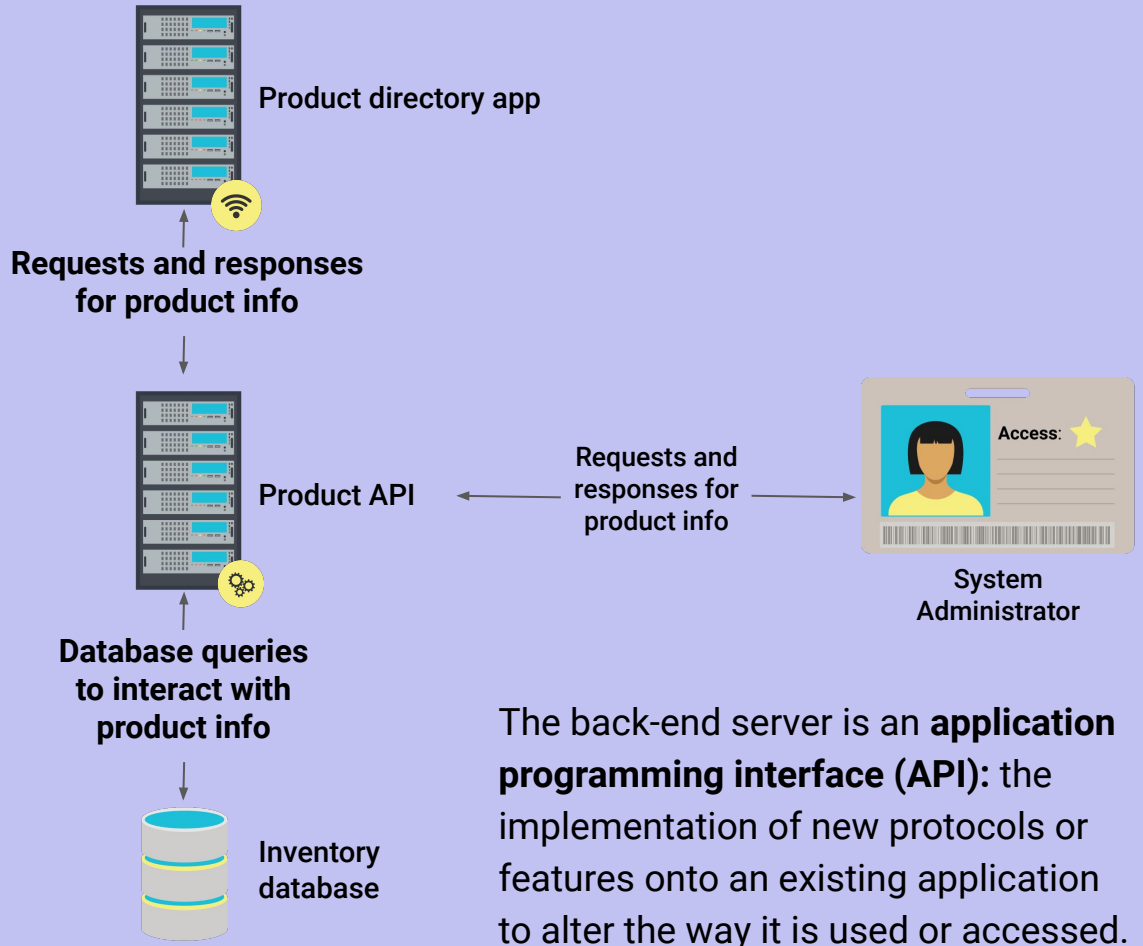
When Amazon customers want to retrieve a list of all the newest and most popular books, they do not go to the back-end MySQL server to check the inventory.

- They go to the front-end microservice.
- Then, the microservice communicates with the back-end server, and brings up the relevant data.
- Customers do not have access to the back-end server. The back end does not accept HTTP requests, meaning users can't communicate with it.

APIs

Systems administrators need to access and query the back end to update product information.

- Since they need to directly communicate with the back end, they must modify it to accept requests.
- Software developers will program the back end to receive, from sources other than the front-end, HTTP requests to add, remove, and view product information.

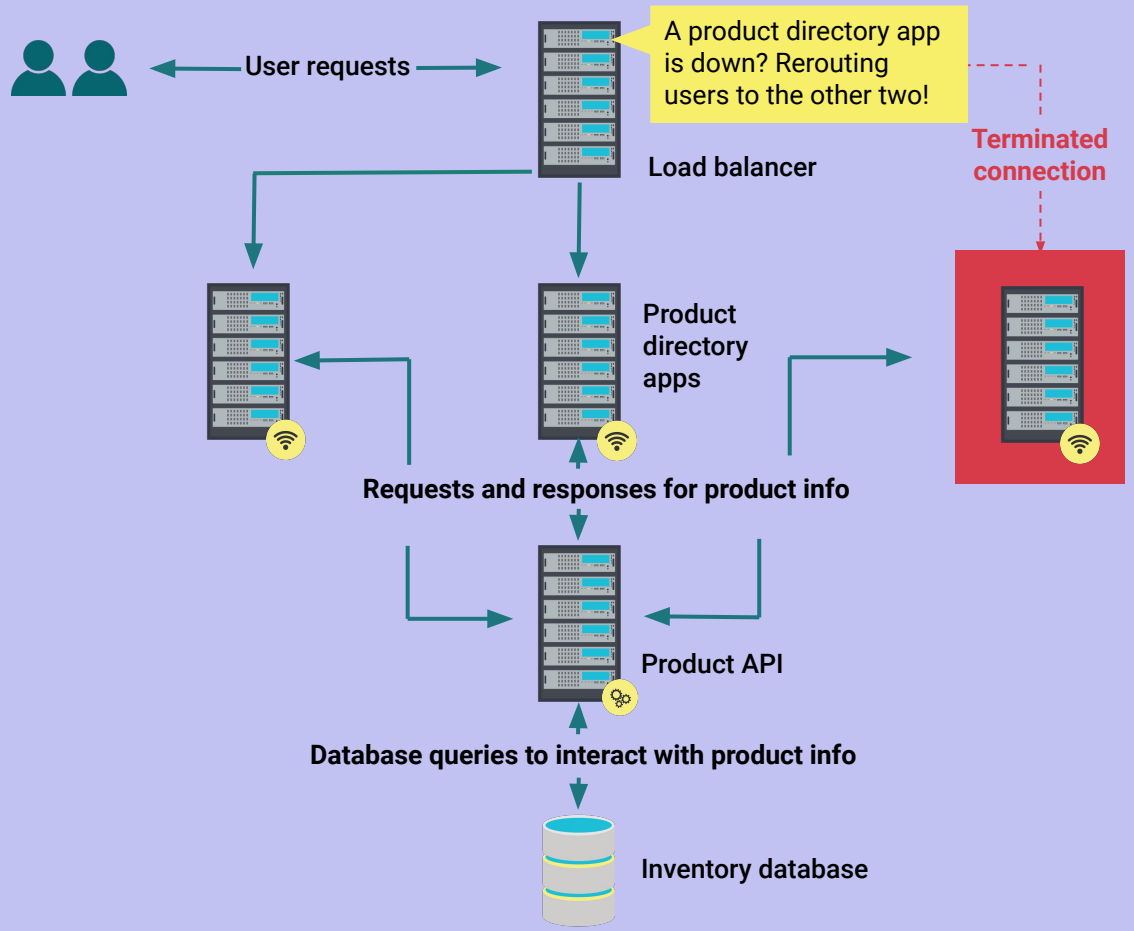


The back-end server is an **application programming interface (API):** the implementation of new protocols or features onto an existing application to alter the way it is used or accessed.

Challenges of Microservices

Microservice complexity and interconnectivity also come with unique security issues.

- Microservices have **increased complexity and require more maintenance** as the application and number of components grows.



Monolith to Microservice

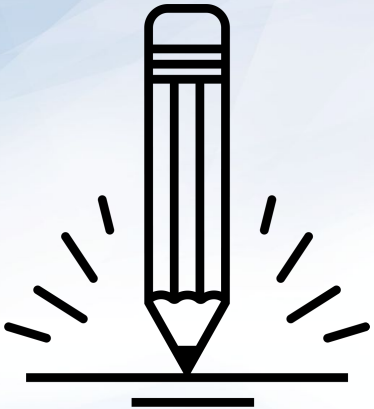
In the next activity, you will diagram microservices.

01 Separate each component of the monolith by its function and move it to its own machine.

02 Add communication between each part of the microservice.

03 Turn the back-end server into an API to interact with more than just the front end.

04 Rename the rest of the component services to match the main function they provide.



Activity: Monolith to Microservice

In this activity, you will diagram a microservice version of a monolith web store application.

Suggested Time:
15 Minutes

Time’s Up! Let’s Review.

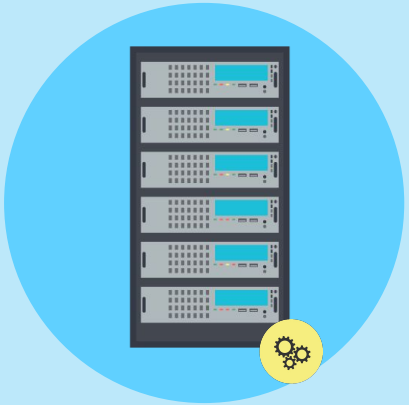
Web Application Architecture

Web Application Architecture

One of the first steps to creating microservices is separating the components into their own individual machines so that each one can run a “service.”

A service, such as the front-end server from our earlier examples, is really just an operating system, running the minimum software that was configured to serve one main purpose, such as handle HTTP requests and responses.

Front-end HTML server



Back-end application server

MySQL database

Web Application Architecture

Each individual machine requires:

An operating system to run on, in order to run the service's software and store its configurations.

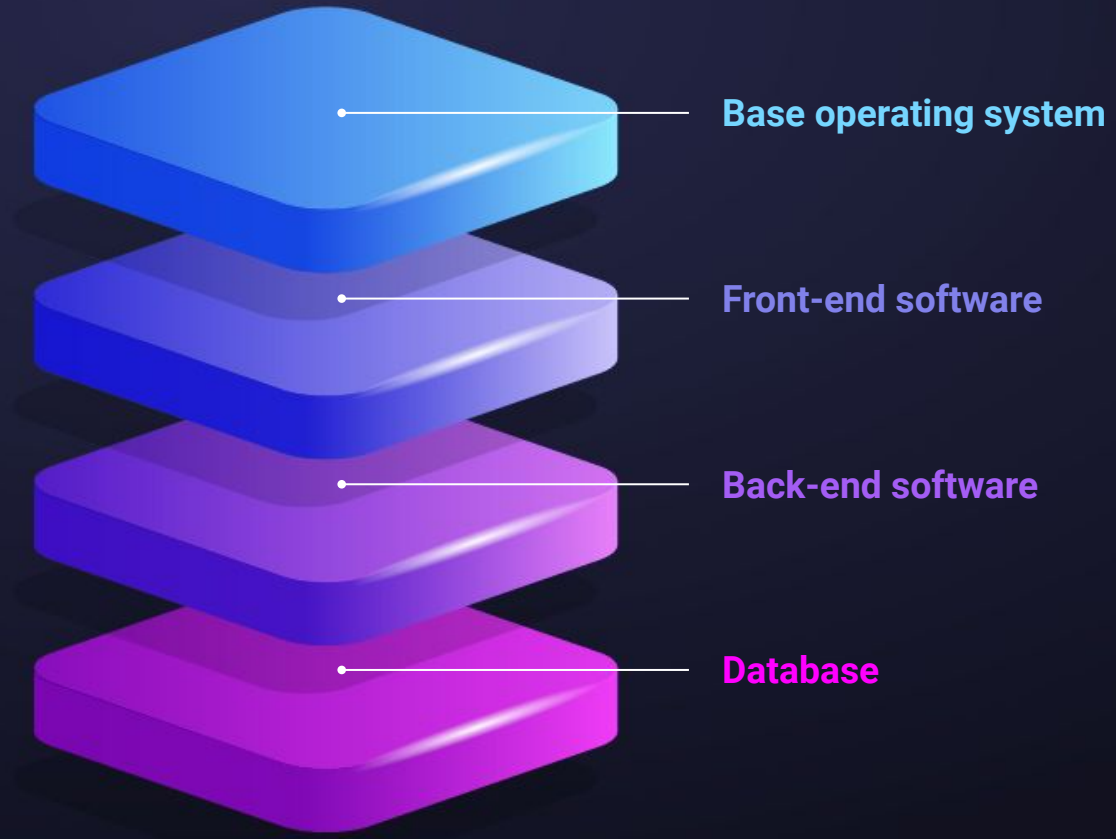
The software that runs a specific set of instructions for that individual component of the web application.

For example: A database service may run Linux as its operating system, and then have Microsoft's SQL Server installed on it.

Introducing Stacks

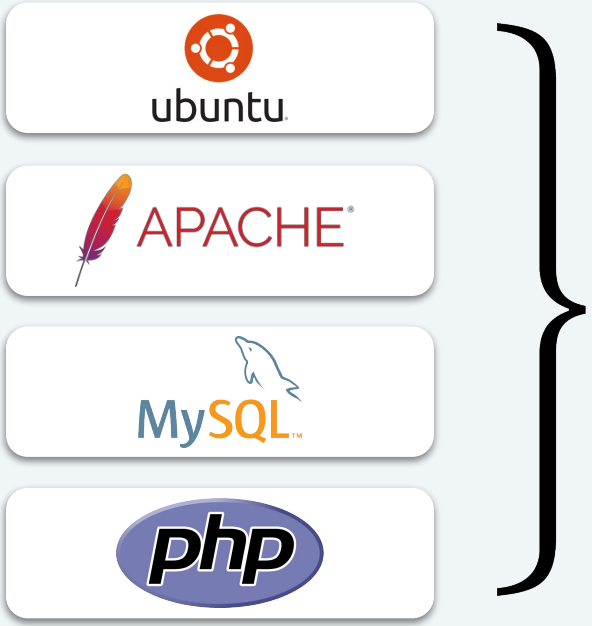
What open-source and commercial software do we need to set up in order to create our web application components?

Any combination of a base operating system, front-end software, back-end software, and a database is called a web **stack**.



LAMP Stack

A popular example of a stack is the **LAMP** stack.



L **Linux,** as the base operating system.

A **Apache,** to handle the frontend HTTP requests/responses.

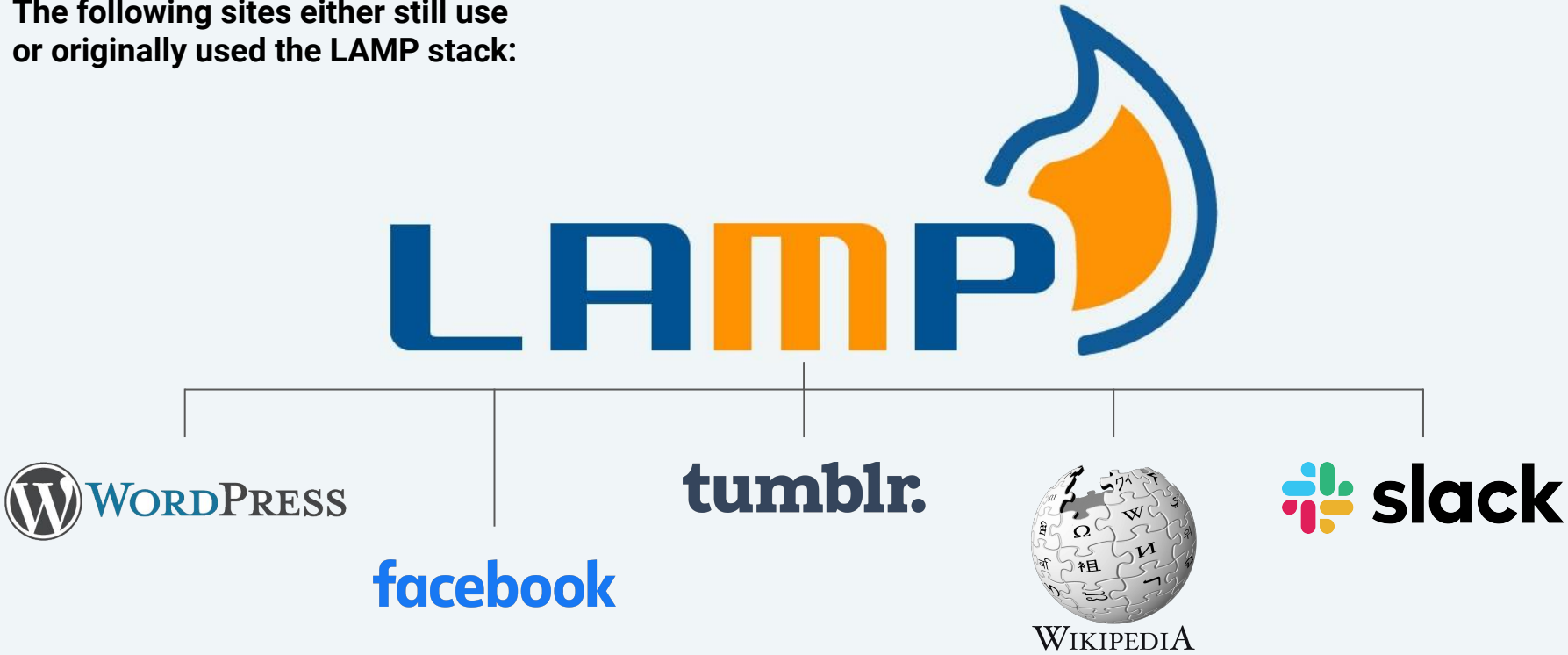
M **MySQL,** for storing data.

P **PHP,** for back-end transformation of front-end requests and responses to database queries.

LAMP Stack

The LAMP stack is considered the tried and true web stack, with a long history of use in architecting web applications.

The following sites either still use or originally used the LAMP stack:



More Stacks

These days, there are many more web stacks, which use different variations of software.

But what stacks should we use for our own web application?

Let’s take a look at some options.

Common Web Architecture: Operating Systems

Let's start by comparing the different **operating systems** available for our services.

The **Alpine Linux** is a super lightweight Linux distribution that has the minimum requirements to run containers.

Other container-friendly operating systems include most other Linux distributions, such as Ubuntu, and more recent versions of Windows Server.

The following is a list of base operating systems used for web applications:

Common Linux distros such as:

ubuntu Red Hat SUSE

Windows Server 2016 and higher

Windows Server 2016

Common Web Architecture: Front-end Server

Our front-end server handles the HTTP requests and responses.

It is usually some form of HTTP server running on a lightweight containerized operating system.

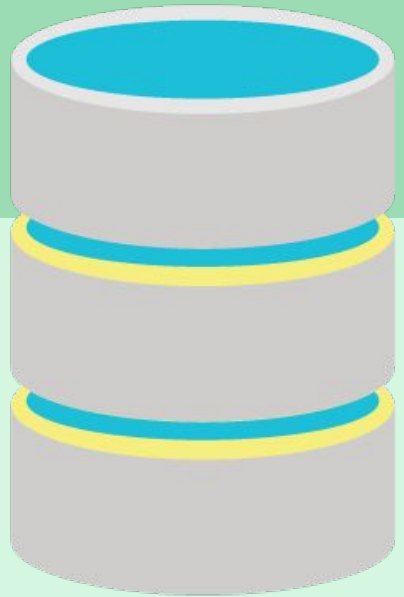
Front-end server is the part of a web application that renders the visual components of a web application. HTML, CSS, and JavaScript are the primary front-end web design languages.

Examples include:

- Apache
- nginx
- IIS

Common Web Architecture: Database Server

The database server is the part of the web application that houses data for the application.



Remember that there can be multiple databases, some for storing sessions and cookies, some for storing business logic like a company's employee directory.

Examples include:

- MySQL
- MariaDB
- PostgreSQL
- Microsoft SQL Server
- CouchDB
- MongoDB

Common Web Architecture: Back-end Server

Lastly, we'll usually need a back-end server that transmits data between the front-end and the database.

These servers run fully-featured scripting and programming languages.

Examples of back-end servers and languages include:

- PHP
- Perl
- Python
- nodeJS

Honorable Mention: The WIMP Stack

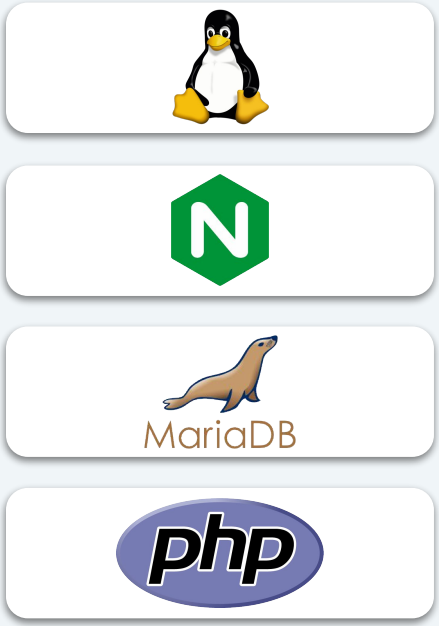
There is also a Microsoft-based variation called the **WIMP** stack that uses the following:

Like the LAMP stack, we won't be using WIMP in class, but it is good to know that Microsoft-based web stacks are available.

- **W** — **W**indows operating system
- **I** — Microsoft's **I**IS web server in place of Apache/nginx
- **M** — A database, such as **M**icrosoft SQL Server
- **P** — Back-end language, such as **P**owerShell

Introducing the LEMP Stack

The web software stack that we'll be using for today's demos and activities consists of:



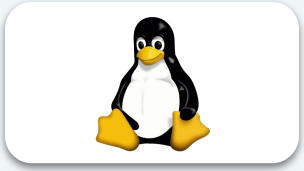
L — **L**inux

E — nginx (pronounced "**e**ngine-x")

M — **M**ariaDB, and

P — **P**HP

LEMP Stack



L

E

M

P

Linux

The standard underlying operating system used for web applications. However, we will be using it with containers.

nginx

We'll be using nginx for our frontend HTTP server, instead of Apache, mainly for its speed. While Apache is considered to have more features and is more compatible with backend languages, such as Java, nginx is well-known for its performance.

MariaDB

Unofficially the "other" MySQL, it is a standard relational database that will handle our data storage.

PHP

Our pages will be generated with data pulled by PHP scripts. We will not need to manage or work with any PHP directly today, but know that it is acting as our back-end language.

Now that we figured out the software, we need to work out a hardware environment suitable for web application microservices that is both lightweight and easy to deploy.

Microservice Infrastructure

Microservices require lightweight environments to run because:

- Live services need to be deployed quickly.
- Multiple copies of a service can be replicated as needed to meet demand.
- Developers and maintainers can deploy their own copies of these services locally for testing purposes.
- Full-sized VMs for each service require more resources and are more expensive.

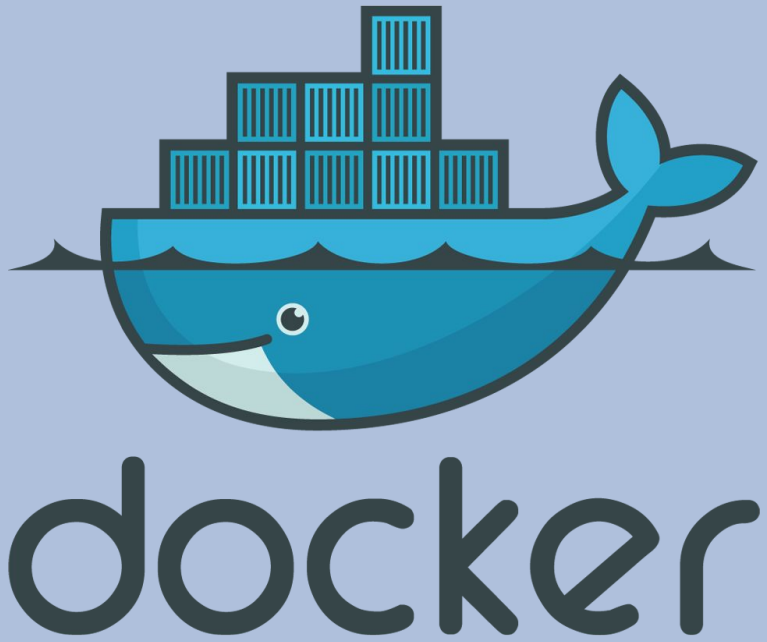
Each service needs its own lightweight virtual environment.

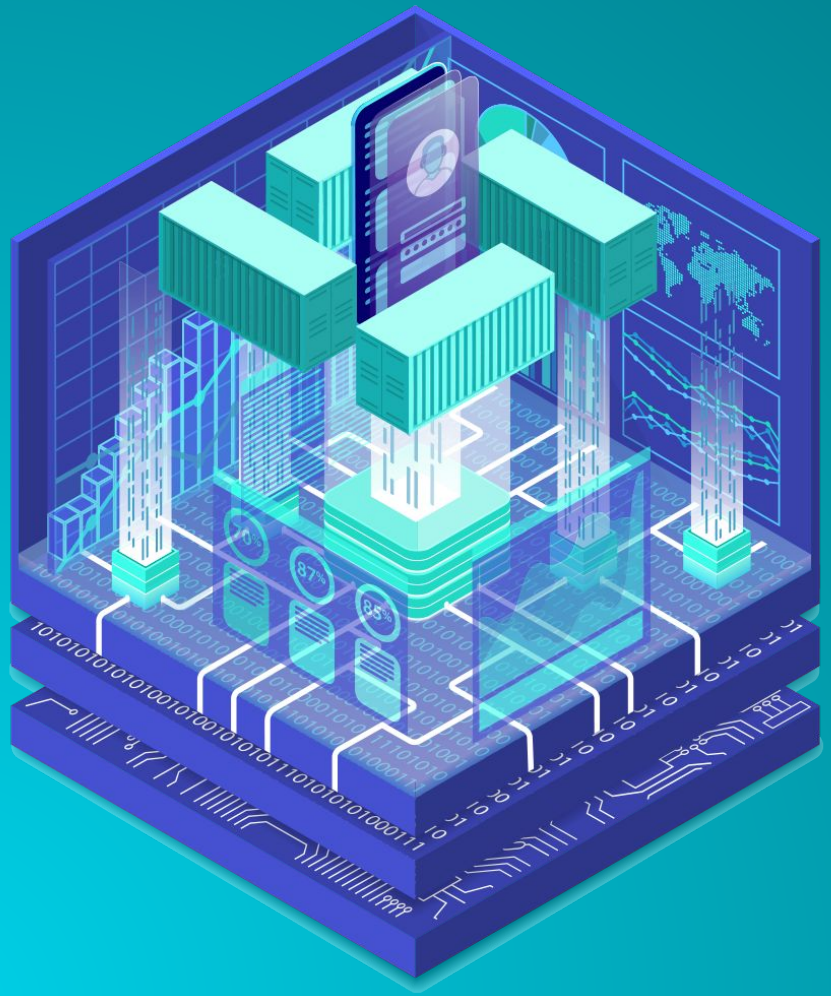
Can we think of any technologies that provide a lightweight, isolated environment?

Microservice Infrastructure with Containers

In the cloud security unit, we covered containers and Docker extensively. We also used Ansible to set up lightweight containers in our activities.

- Containers run as isolated, virtual operating systems that dynamically allocate resources depending on needs. This is unlike regular virtual machines, where all hardware is virtualized.
- Docker is the most popular container platform. For the rest of this class, we will be referring to Docker's implementations of containers.

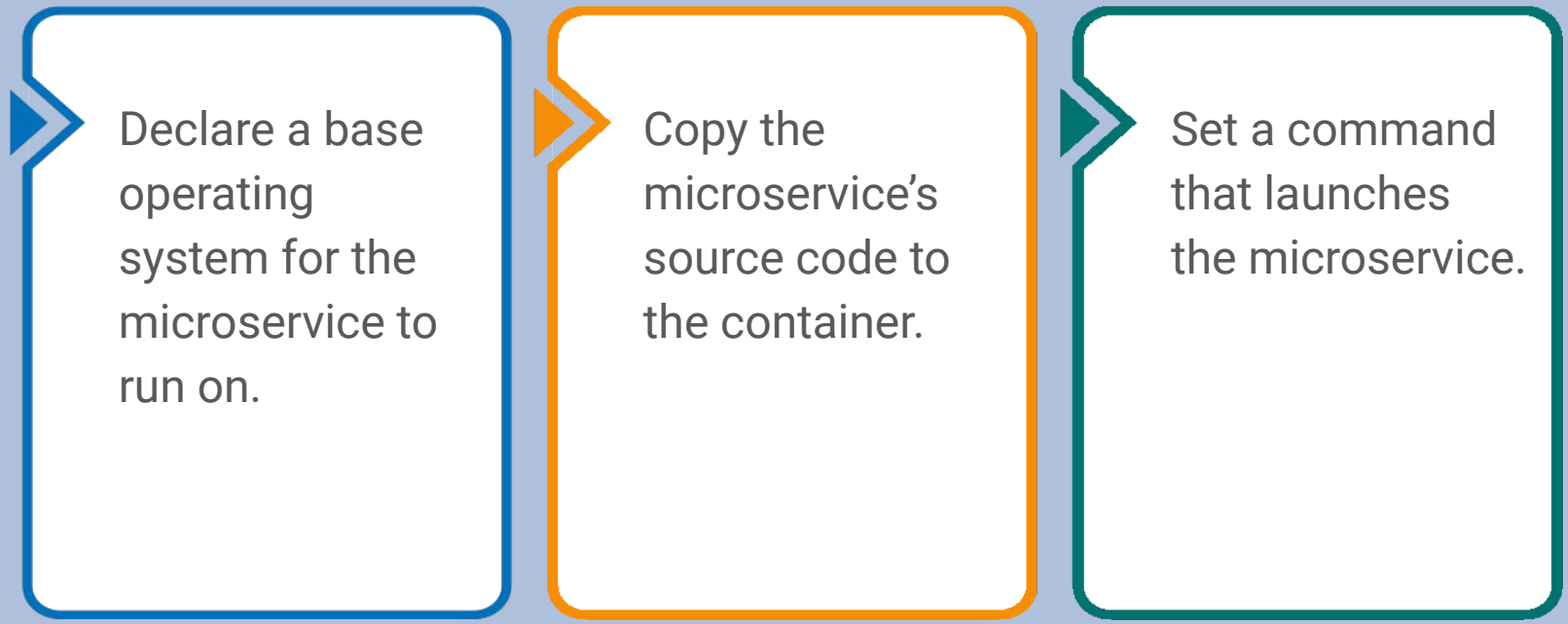




Containerization is the process of packaging all the requirements for a microservice into a container.

How a Container Becomes a Microservice

Containerizing a microservice requires the following steps:



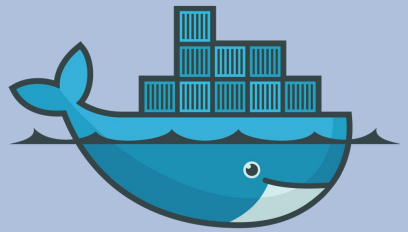
Dockerfile

Containerization can be declared in a text file called a **Dockerfile**.

```
###################################
# Employee Directory Application
###################################
# Required Base OS
FROM ubuntu:14.04
  ...[truncated]
```

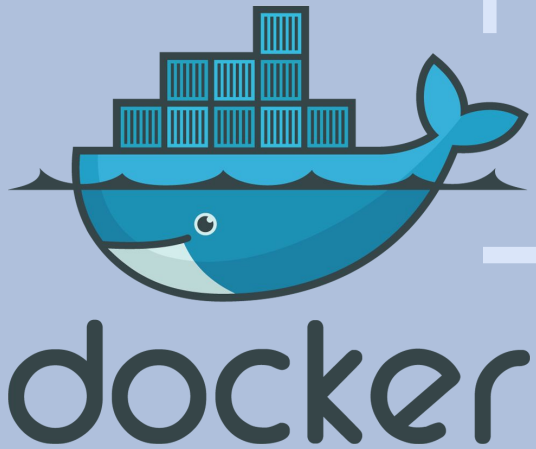
Dockerfiles contain all the configuration needed for a container in one file.

This is similar to an Ansible configuration file, which contains everything needed to configure a host with the command `ansible-playbook your-playbook.yml`.



Dockerfile

A unique Dockerfile is created for each microservice.



Product Directory app

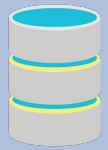
Dockerfile

```
#######################
# Product Directory Application
#######################
# Required Base OS
FROM ubuntu:14.04
```

Product API

Dockerfile

```
#######################
# PRODUCT API
#######################
# Required Base OS
FROM fedora:31
```



Amazon database

Dockerfile

```
#######################
# AMAZON Database
#######################
# Required Base OS
FROM centos:8
```

Deploying and Testing a Container Set

Deploying at Scale

Deploying containers can be tedious, especially if a business requires hundreds or thousands of microservices. Therefore we need to know how to deploy in large-scale environments. Specifically, how to:

- Deploy multiple types of containers at the same time.

  For example, front-end and database services.

- Deploy copies of containers we want replicated.

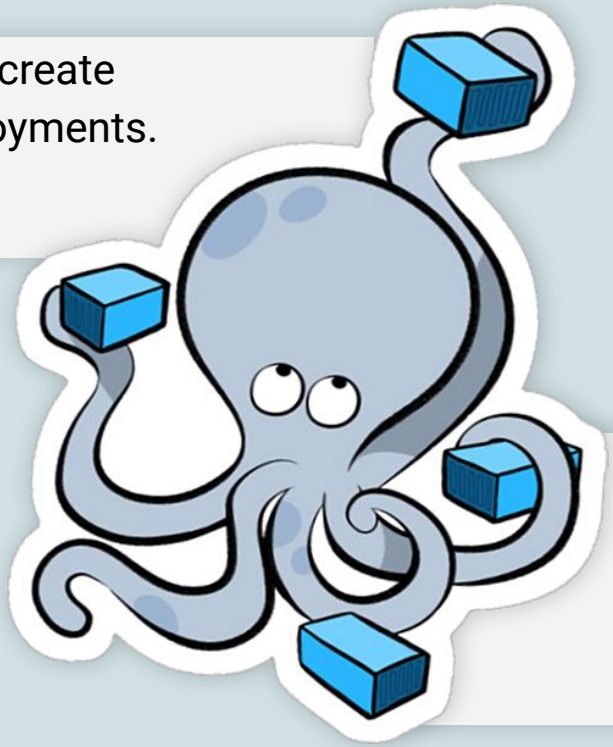
- Set up a network for our containers so they can communicate with each other.

Deploying at Scale

We'll use Docker Compose to enable these large-scale deployments.

Docker Compose allows us to create repeated, multi-container deployments.

docker compose



Remember: We used Docker Compose in the last class to set up our WordPress site.

Docker Compose YAML File

We also used YAML files to deploy Ansible playbooks during our cloud security unit.

- Docker Compose uses YAML to define the containers, their networking configurations, and where you want to copy files from your host machine into your container.

```
version: "3.7"

services:
  ui:
    image: httpd:2.4
    ports:
      - 10000:8080
    volumes:
      - ./volume:/home
    networks:
      demo-net:
        ipv4_address: 192.168.1.2

[truncated…]
```

Docker Compose Demo

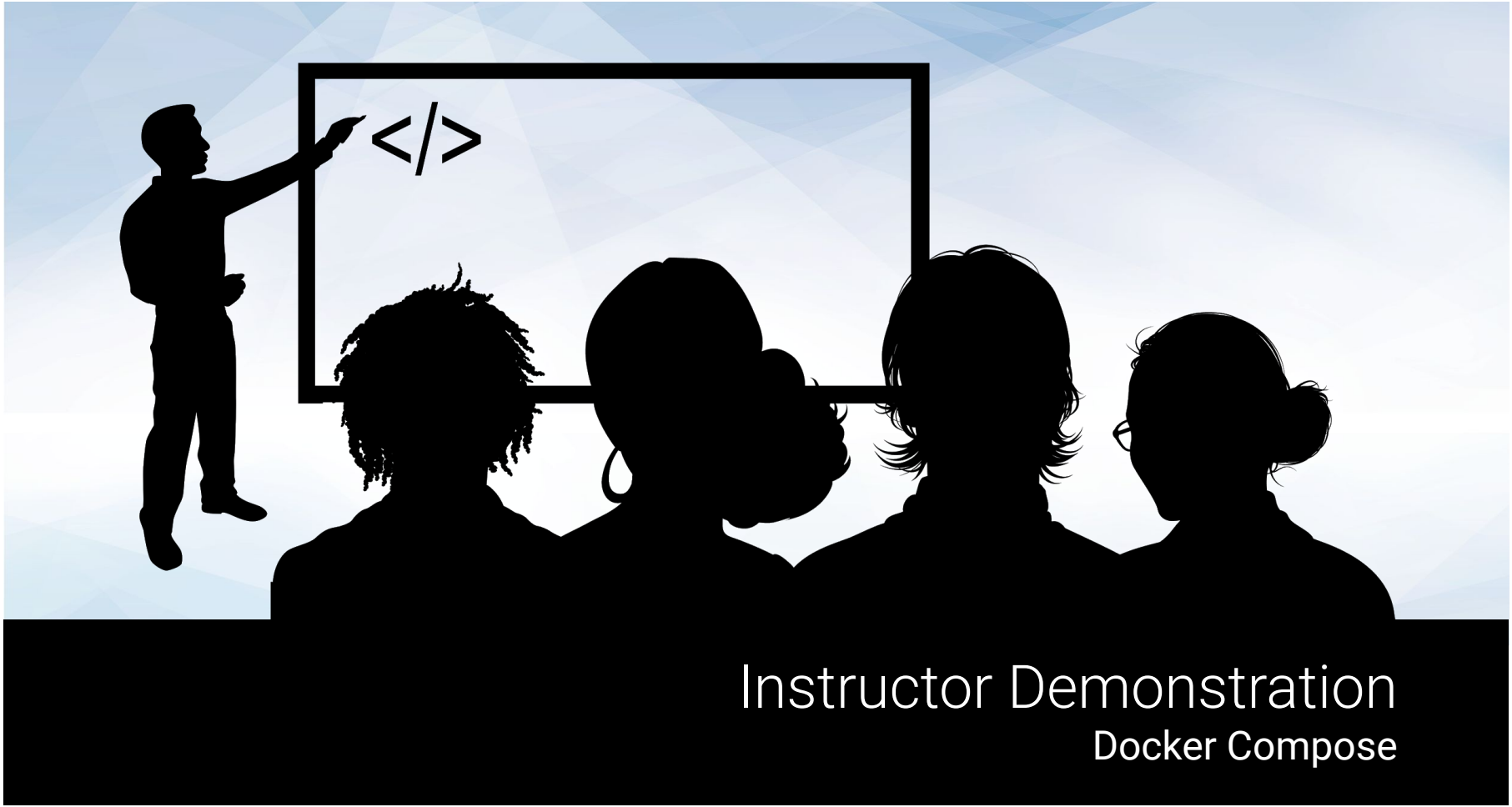
For this web application deployment demo, we'll complete the following steps:

01 Examine the YAML file for its current configurations.

02 Launch our services with `sudo docker-compose up`.

03 Use our browser to verify the our front-end user interface (`ui`) services are deployed properly.

04 Enter a MySQL session to confirm proper deployment of our database service.



Instructor Demonstration

Docker Compose

```yaml
version: "3.3"

services:
  ui:
    image: httpd:2.4
    ports:
      - 10000:8080
    volumes:
      - ./volume:/home
    networks:
      demo-net:
        ipv4_address: 192.168.1.2
  db:
    image: mariadb:10.5.1
    restart: always
    environment:
      MYSQL_DATABASE: demodb
      MYSQL_USER: demouser
      MYSQL_PASSWORD: demopass
      MYSQL_RANDOM_ROOT_PASSWORD: "1"
    volumes:
      - db:/var/lib/mysql
    networks:
      demo-net:
        ipv4_address: 192.168.1.3
networks:
  demo-net:
    ipam:
      driver: default
      config:
        - subnet: "192.168.1.0/24"
volumes:
  ui:
  db:
```

```
services:
  ui:
  db:
```

We'll break down the file by the services it installs, the user interface, and the database.

The `ui` and `db` are services.

```
ui:
    image: httpd:2.4
    ports:
      - 10001:8080
    volumes:
      - ./volume:/home
    networks:
      demo-net:
        ipv4_address: 192.168.1.2
```

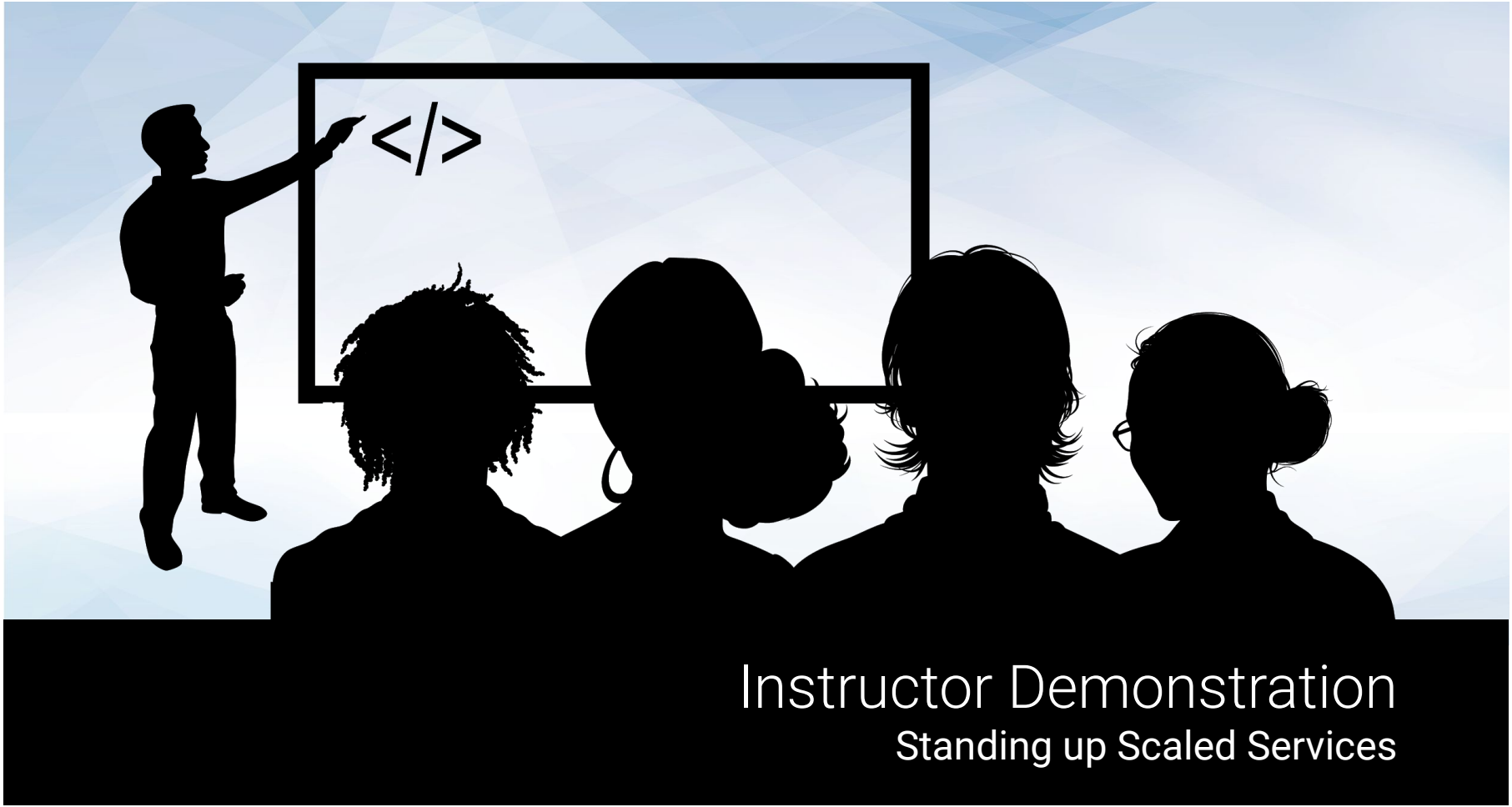
image: Points to where Docker Compose will retrieve the container image on Docker Hub, which is Apache's httpd container.

Ports: Ports this container will run on.

volumes: Local destination where the Apache server will save configuration files.

networks: The network that this service will connect to.

Ipv4_address: The static IP address we are assigning to this container.



Instructor Demonstration

Standing up Scaled Services

```yaml
db:
  container_name: demo-db
  image: mariadb:10.5.1
  environment:
    MYSQL_DATABASE: demodb
    MYSQL_USER: demouser
    MYSQL_PASSWORD: demopass
    MYSQL_RANDOM_ROOT_PASSWORD: "1"
  volumes:
    - db:/var/lib/mysql
  networks:
    demo-net:
      ipv4_address: 192.168.1.3
```

We can find declared database credentials in the YAML file under the environment sub-heading.

`image:` The container image and version we'll be using. (MariaDB database version 10.5.1)

`environment`: Containers within the MySQL server.

`MYSQL_DATABASE`: The name of the database.

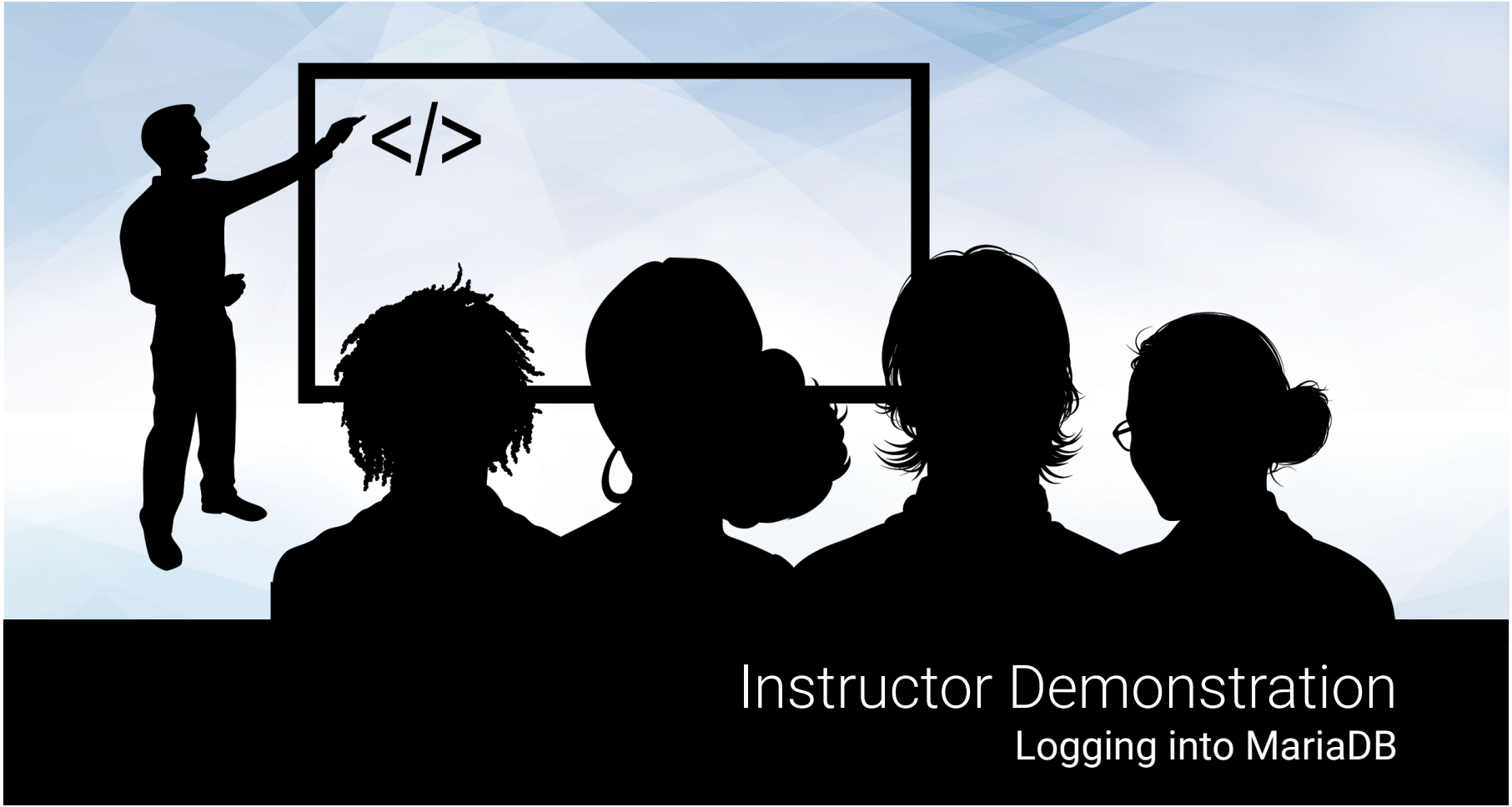
`MYSQL_USER`: The default user for the database.

`MYSQL_PASSWORD`: The password for the user.

`MYSQL_RANDOM_ROOT_PASSWORD:` Gives the root user a random password, for security purposes.

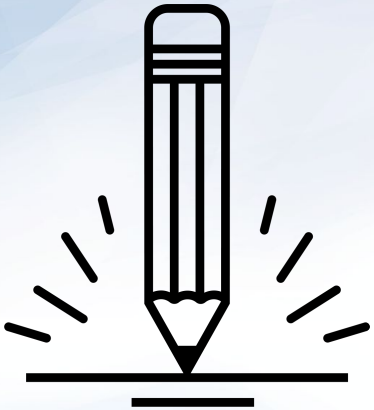
`volumes`: The location where we are saving our configuration files.

`networks:` Assigns this database a static IP address using the same network (`demo-net`).



Instructor Demonstration

Logging into MariaDB



Activity: Deploying and Testing a Container Set

In this activity, you will deploy multiple containers with Docker Compose and test functionality.

Suggested Time:
15 Minutes


Break

Time’s Up! Let’s Review.

Introduction to Databases

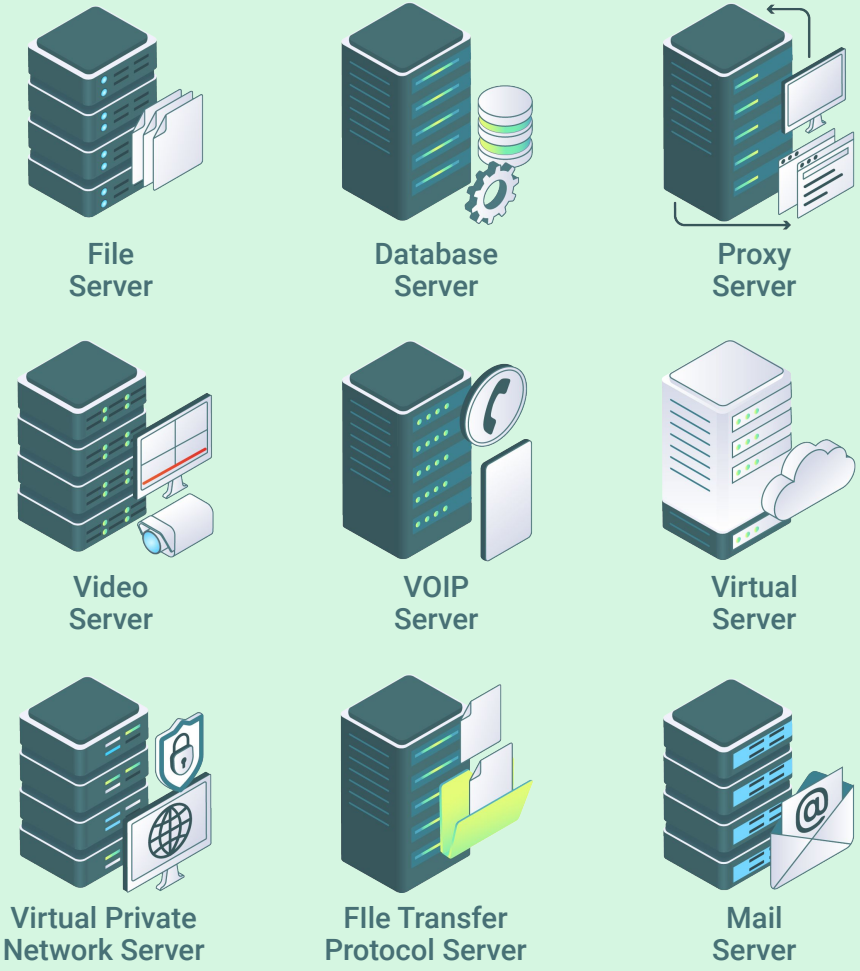
Now that we have easily deployable microservice-based web applications, we'll cover one of the most critical components of a web application: the database.

The Importance of Databases

Databases are so critical to web application architecture because they contain the persistent data that flows through the web application.

The data in the database can include, but isn't limited to:

- Usernames, telephone numbers, and email addresses of customers.
- Session and cookie IDs of site visitors.
- Personally identifiable information (PII), such as the full names, dates of birth, credit card numbers, and social security numbers of customers.



The Importance of Databases

It is absolutely crucial that security professionals have a strong understanding of how databases work in web applications.

We'll take a closer look at the threats and vulnerabilities of databases next unit.

Databases Overview

Databases are used to store large amounts of data.

To assess threats and mitigate risks, we need to look at each component of an organization, and understand how malicious actors can exploit weaknesses and damage the stakeholders' finances, reputations, and well-being.

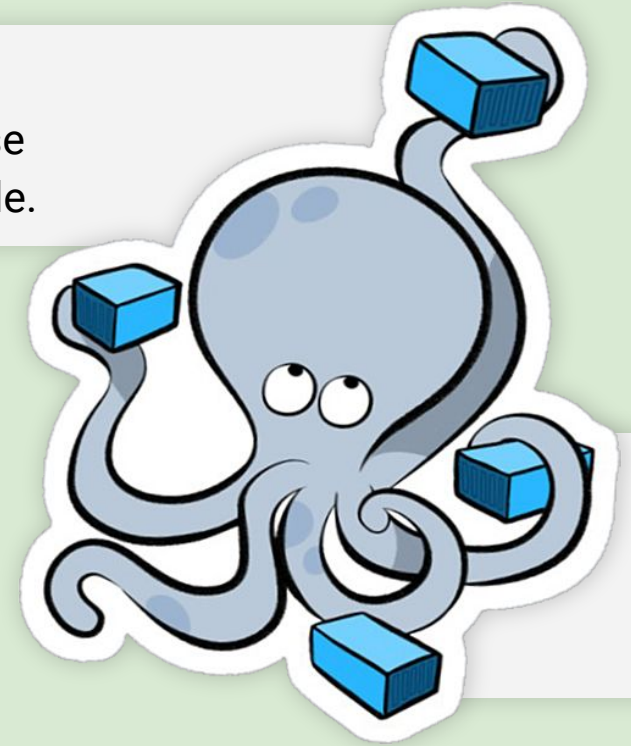
Database Overview

Databases are also typically kept separate from the other components of a web application.

Why we separate databases from other web apps:

Security	Compromise of one machine does not imply compromise of the other.
Scale	Many servers may use the same database.

As we saw in our last demo and activity, this was the case in our `docker-compose.yml` file.



We had multiple defined services, and the database was its own container `demo-db` in the demo and `db` in the activity.

SQL Databases

While the two most important types of databases are SQL and NoSQL, we will be focusing only on SQL today.

SQL databases organize data like an spreadsheet.

Each **column** is a piece of data in the row.

Each **row** is an item in the database.

ID	First Name	Last Name	Department
1	Jane	Doe	Security
2	Bob	Competent	Marketing
3	Jon	Rolls	Modeling

A whole “spreadsheet” of rows and columns is called a **table**.

A collection of tables is called a **database**.

Interacting with Data

Once you've created a table, there are four main ways to interact with the data it contains.

Create **R**ead **U**pdate **D**elete

CRUD

Interacting with Data

Once you've created a table, there are four main ways to interact with the data it contains.

Create

or add new entries to a database.

When a user makes a new account in a web app, they are creating a new database entry with their user info.

Interacting with Data

Once you've created a table, there are four main ways to interact with the data it contains.

Read

or view entries in a database.

When a web application verifies your login credentials, it is reading the credentials saved in the database.

Interacting with Data

Once you've created a table, there are four main ways to interact with the data it contains.

Update

or modify existing data in a database.

When you need to reset your password in a web application, the password entry (more specifically, the hashed password) is updated with the new password.

Interacting with Data

Once you've created a table, there are four main ways to interact with the data it contains.

Delete

or destroy data in the database.

If you have the option to delete your account from a web application (for example, on a GDPR web app), your account info is deleted from the database.

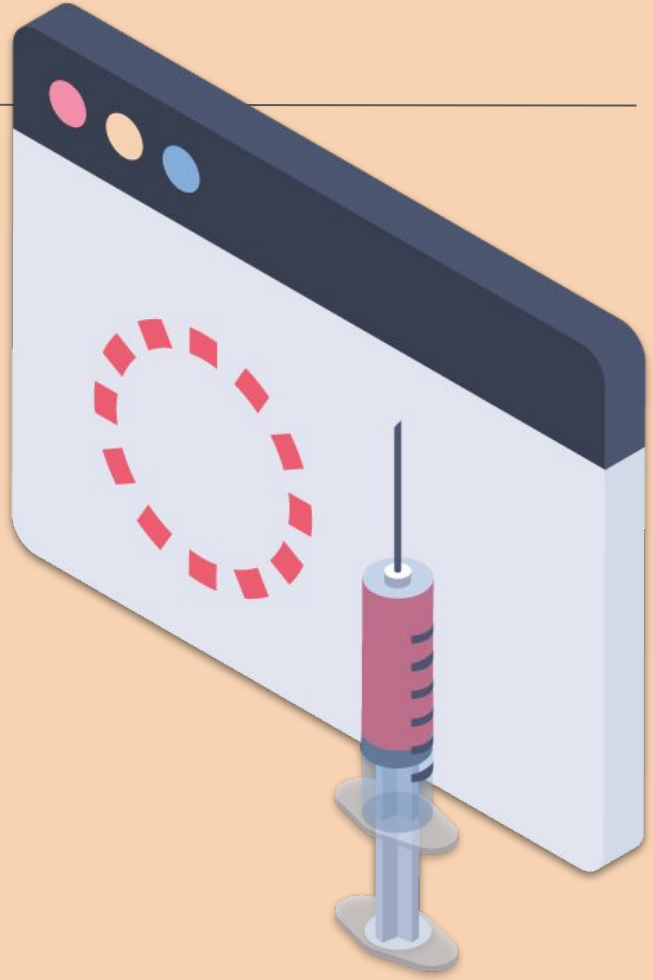
CRUD

Together, these are as known as CRUD operations. These operations are also known as **queries**.

We'll use the term “query” to describe any input that will interact with data.

Next unit, we’ll come back to the the SQL syntax of these operations to set the stage for SQL injections.

SQL injections are a type of web application attack intended to manipulate queries so that an attacker can use a database in ways the developers did not originally intend.



Now that we understand high-level database operations, let’s see how we can create and enter queries into a web application's database to carry out basic database administration tasks.

Database Demo

In this demo, we'll complete database administration tasks:

01 Deploy a web application.

02 Navigate to the web application's page to see what data can be viewed by a web client.

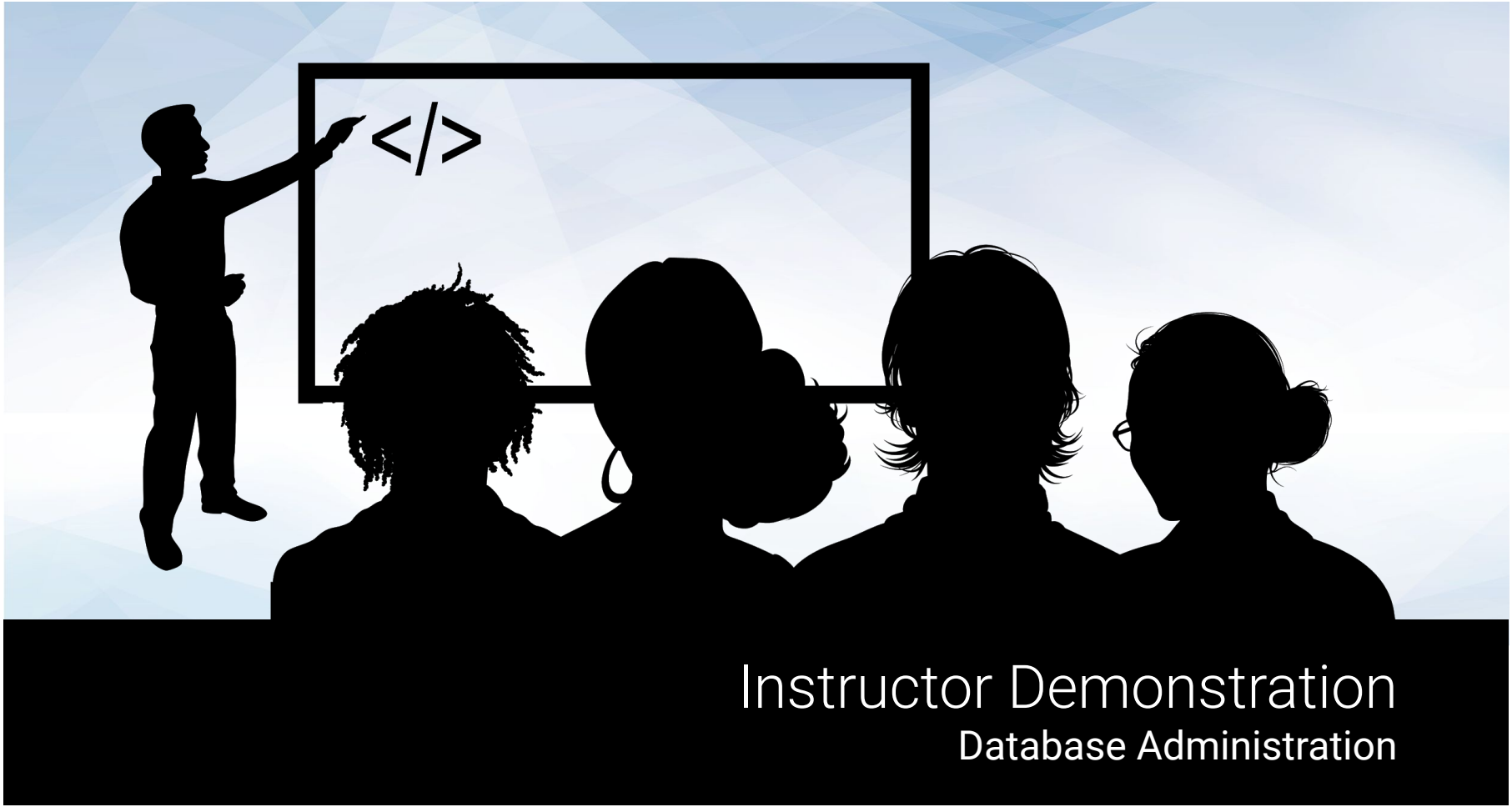
03 Start a bash session in the database's container and log into the database.

04 Use an SQL query to retrieve the contents of the database's primary table.

05 Modify that SQL query to find specific data.

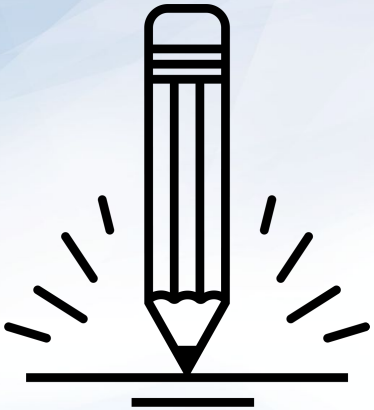
06 Add new data to the database.

07 Delete data from the database.



Instructor Demonstration

Database Administration



Activity: Database Management

In this activity, you will locally deploy GoodCorp's employee directory website with Docker Compose and manage the data inside the database.

Suggested Time:
25 Minutes

Time’s Up! Let’s Review.

Next Week

In Unit 15, we will use the Web Vulnerabilities Azure Lab environment.

Inside the Web Vulnerabilities environment are two nested VMs: a Kali Linux instance and a machine called owaspbwa.

- The owaspbwa machine hosts the vulnerable web applications that we will use throughout the class, but we do not log into it.
- Instead, we will perform all of our work on the Kali Linux instance.

Student credentials for the Kali machine:

- Username: `sysadmin`
- Password: `cybersecurity`

Take a moment to set up and access this environment. Troubleshoot any issues before class next week.


the End