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.

Create SQL queries to view, enter, and delete data.

Application Structure

Application Structure

In today's class, we'll take a closer look at the infrastructure behind web applications. Specifically, we'll look at:

- How modern web architecture has evolved to what is known as microservices
- What components make up microservices and their specific software "stacks"
- How web application components can be deployed using containers
- How databases work to understand how web applications store and transmit data

Application Structure

We'll begin with how modern web applications ended up adopting the **microservices model**.

In the Web Vulnerabilities unit, we'll use this foundational knowledge of web application architecture and databases to examine how web attacks like SQL injections can lead to information breaches.

Microservices Overview

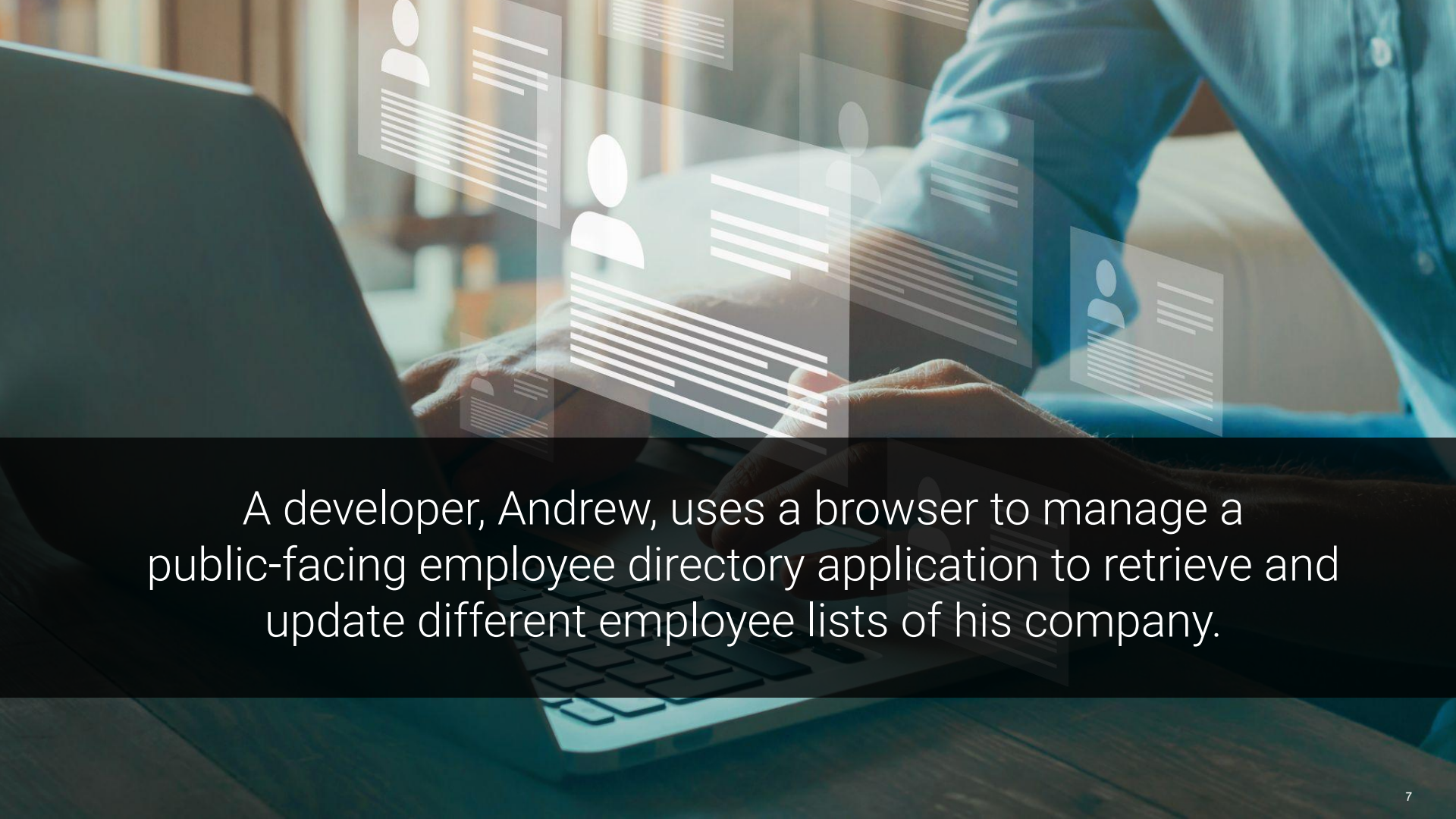
To understand why microservices exist, we need to understand the following:

How an application and its components are organized

How information flows between components of an application

How the application’s architecture has changed from monoliths to microservices

How to deconstruct a monolith into microservices



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

Components of a Typical Web App

The employee directory app is set up so that Andrew can add, remove, and display employees. This application needs the following components in order to function:

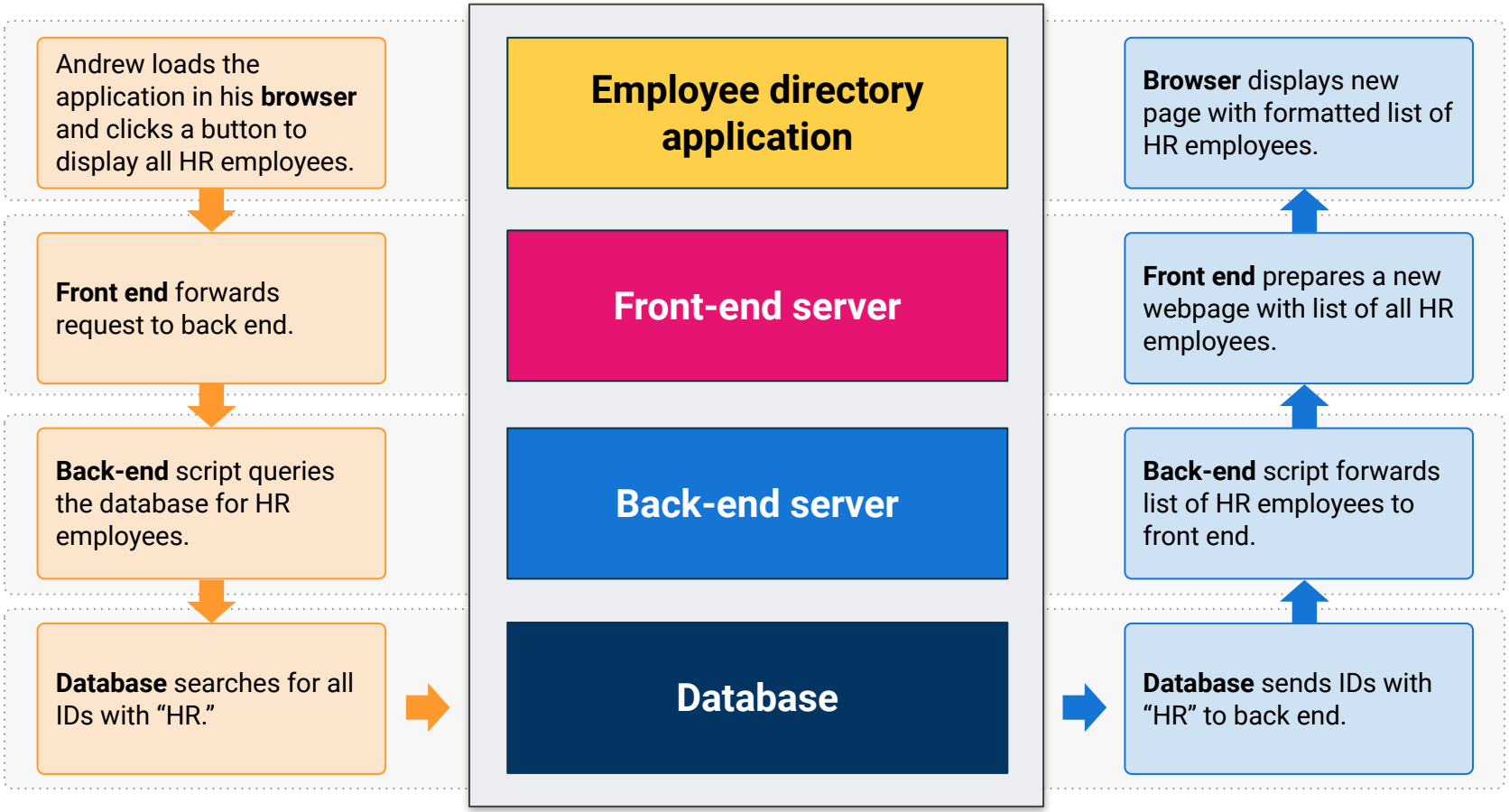
Front-end server	Responsible for displaying webpages and styling them in a readable format. This server also receives and responds to HTTP requests.
Back-end server	Executes business logic and writing or reading corresponding data to and from a database. The back-end server knows how to interact with the database depending on the specific request received.
Database	Stores information on employees, such as their employee IDs and names. Critical and sensitive information is found in databases.

Application components are key to the modern architectural paradigm of microservices.

First, we'll explore monolith architecture and the issues that led to the adoption of microservices.

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.

Information Flows Between Application Components



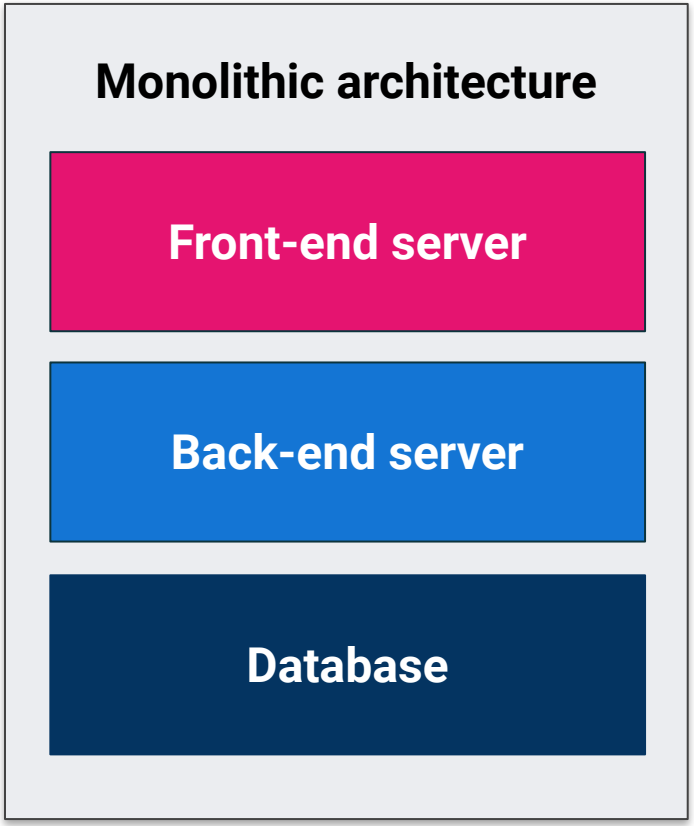
Monolith to Microservices

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

Monolith

In other words…

A monolith is a machine that has a front-end server, back-end server, and database.



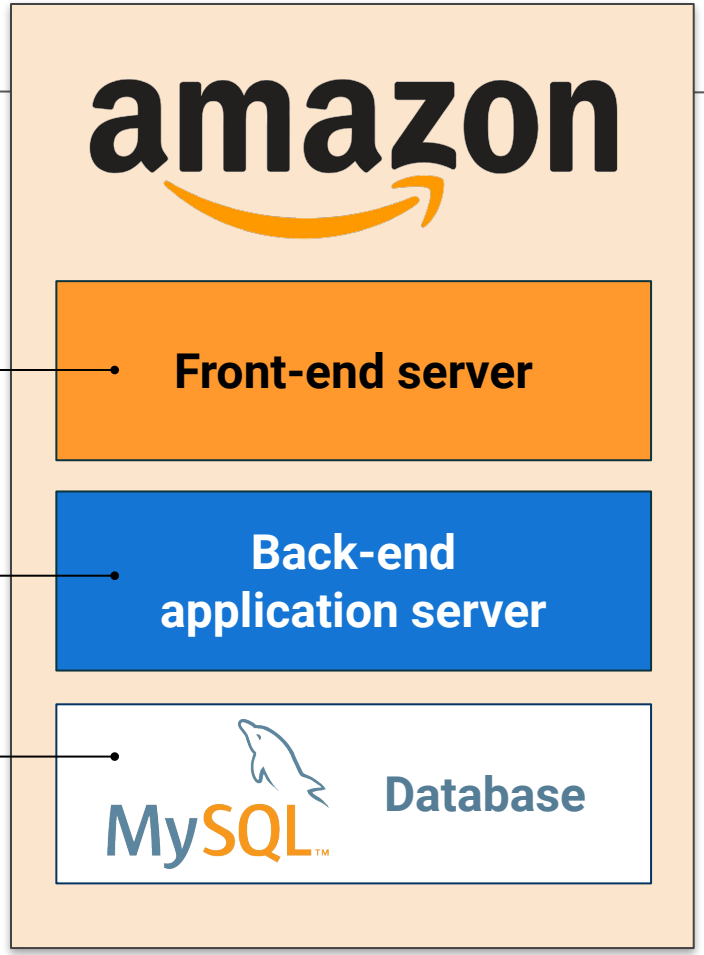
Monolith

Suppose Amazon used a monolithic server. It would contain:

GUI for customers to look at while shopping

Back-end server showing the inventory and stock. Interacts directly with the database

Database of customers and their information and purchases



Problems with Monoliths

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

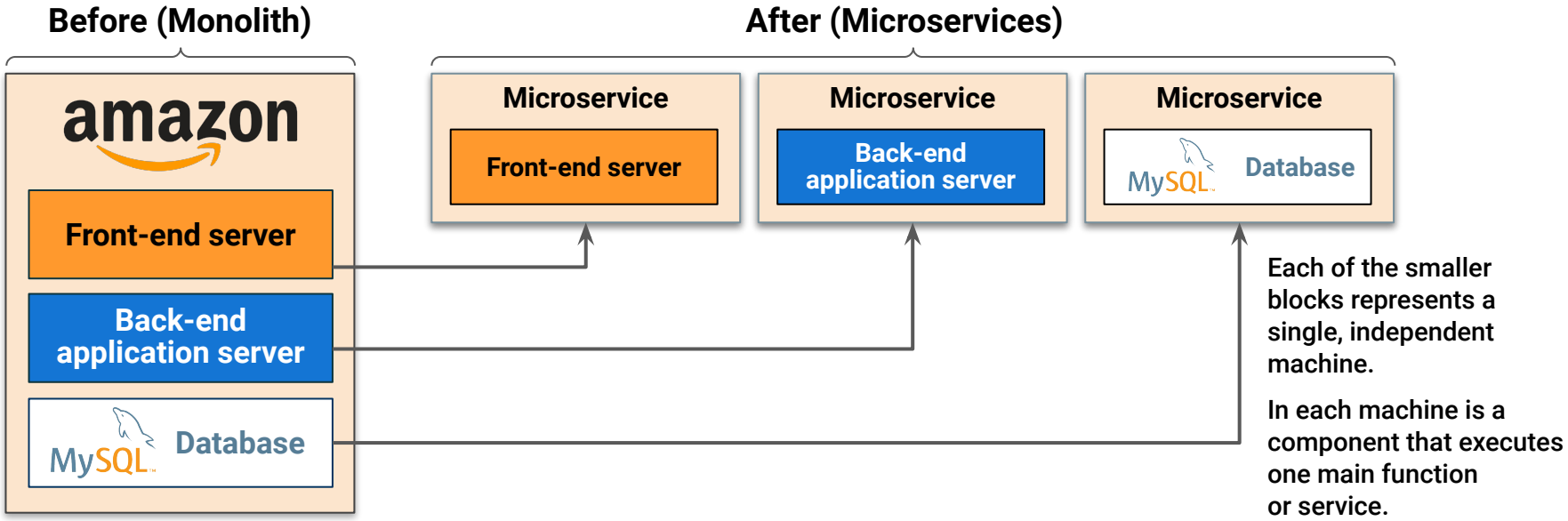
This presents a few problems:

- Updating a single component—the front end, back end, or database—means taking down all components, resulting in downtime and lack of availability.
- Hackers, environmental issues, and human errors that compromise one component threaten the entire machine.

Separating application components into their own machines is called **microservices**.

Microservices

Amazon decides that the monolith architecture comes with too much risk and instead creates a more **modular** setup (a whole consisting of smaller and separate parts) so that system administrators and developers can deploy changes without taking entire applications down for extended periods of time.



Benefits of Microservices

Amazon needs to separate its components into different machines, so that:

- System administrators and developers can deploy changes without taking entire applications down for extended periods of time.
- Separating an application’s components into their own machines allows the sysadmin to update the front end, reboot it, and only have to verify that the front-end components are working.
- They don't have to worry about the back-end components and database.
- If a component is compromised by a hacker, the potential damage is restricted to only that one component.

Benefits of Microservices

Scalability and resilience:

Replication of identical components allows you to serve more clients and provides identical backup components if one fails.

Isolated improvement:

Since microservices should be reduced to serving one primary function, they can be developed to optimize their functionality.

Rapid response:

Since microservice components are inherently smaller than monoliths, they can be replaced and updated quickly.

Isolated security:

One compromised component does not equal a compromised application.

APIs

API Example

When an Amazon customer wants to retrieve a list of the newest and most popular books, they don't go to the back-end MySQL server to check the inventory.

Customers use the front-end microservice.

The microservice then communicates with the back-end server containing that data and brings it up.

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.

API Example

However, Amazon system administrators need to access and query the back end to update product information.

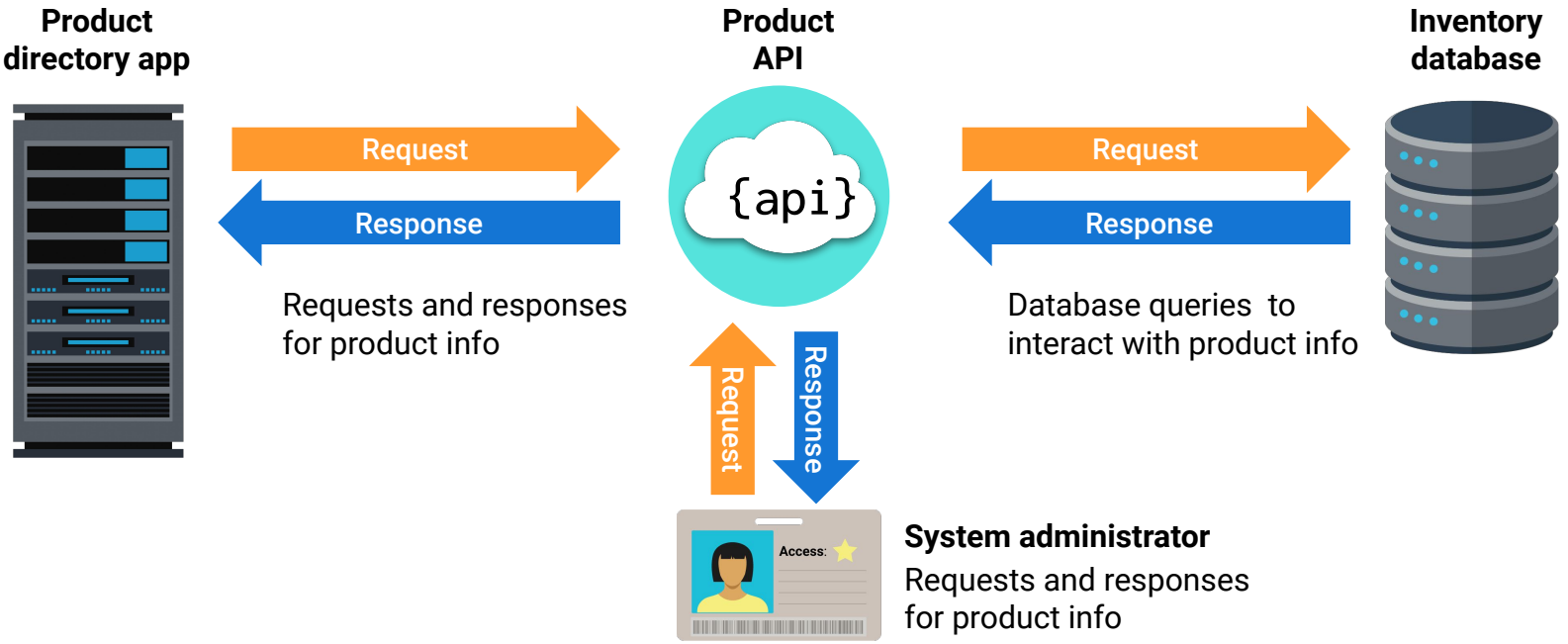
Since they need to directly communicate with the back end, they must modify it to accept requests.

Amazon software developers program the back end to receive—from sources other than the front-end server—HTTP requests to add, remove, and view product information.

An **API** is the implementation of functions/features (App), which make themselves available for Programmatic interaction using a contractual standard Interface.

API Example

The back-end server can now process HTTP requests and responses for product info. However, the front-end server still sends requests and responses for customers requesting product info over the open web as usual.




Recap

Monolith to Microservice

How we separated a monolith into microservices:

01 Separate each component of the monolith by function, moving it into its own machine.

02 Add communication between each microservice.

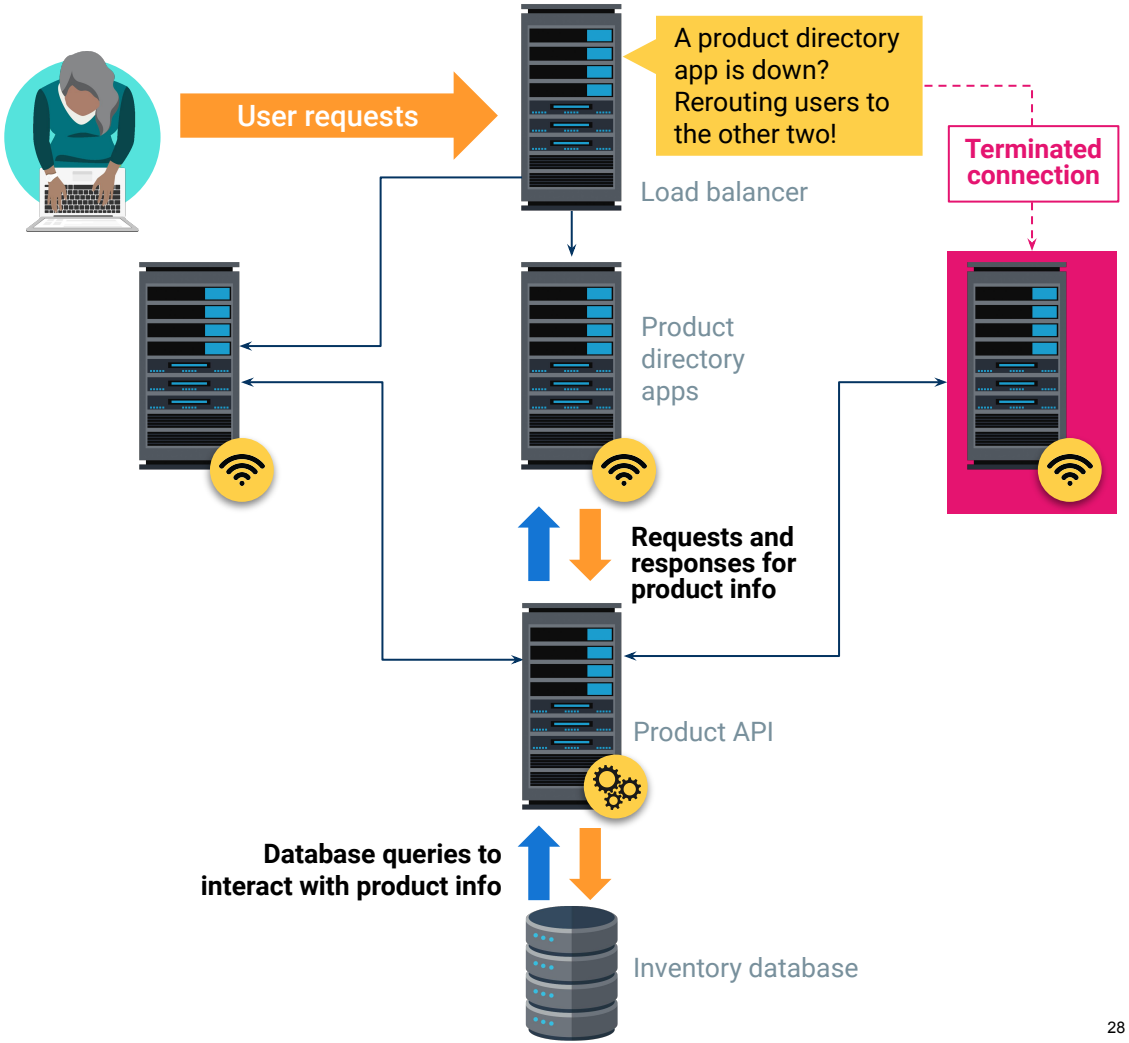
03 Turn the back-end server into an API to interact with more than just the front end.

04 Rename the rest of the component services to match their main functions.

Challenges of Microservices

Microservice complexity and interconnectivity also come with unique security issues.

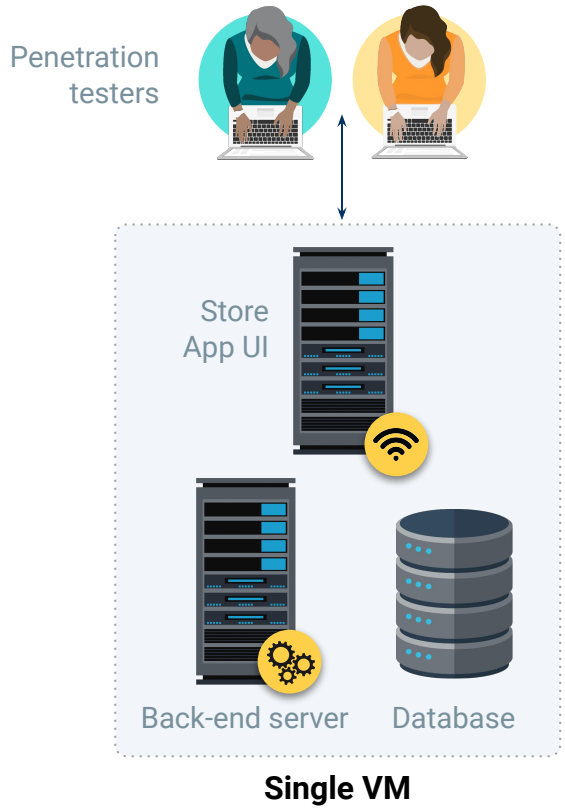
Microservices have **increased complexity and require more maintenance** as the application and number of components grows.



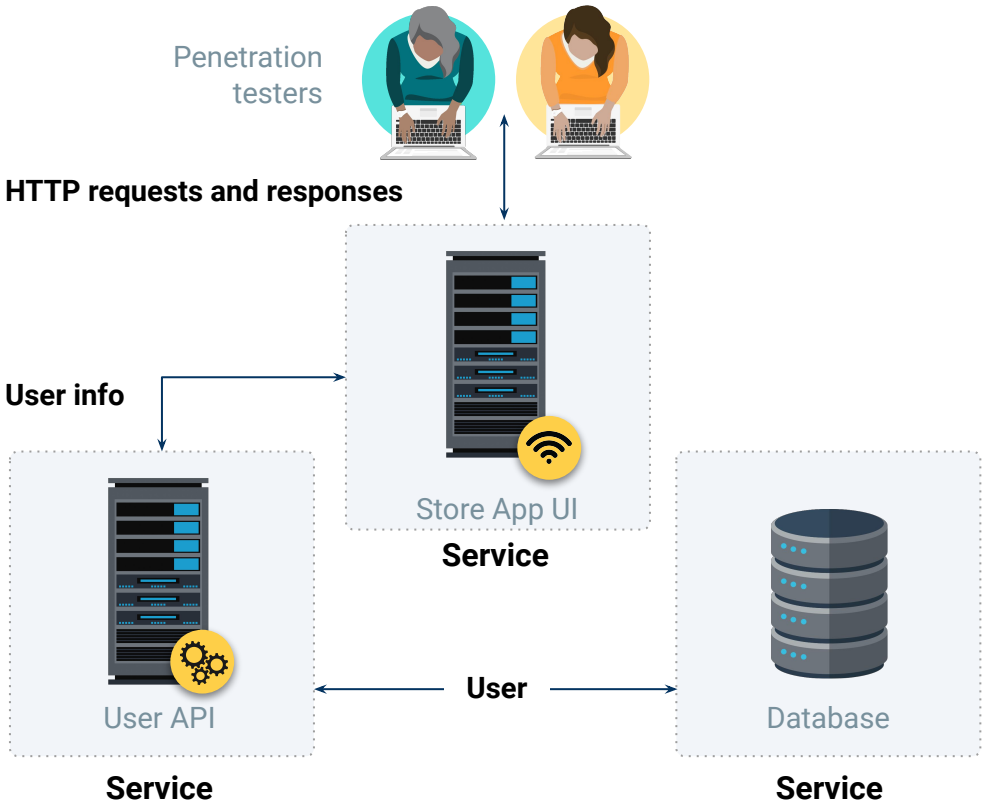
Diagramming Microservices

Diagramming Microservices

Monolith

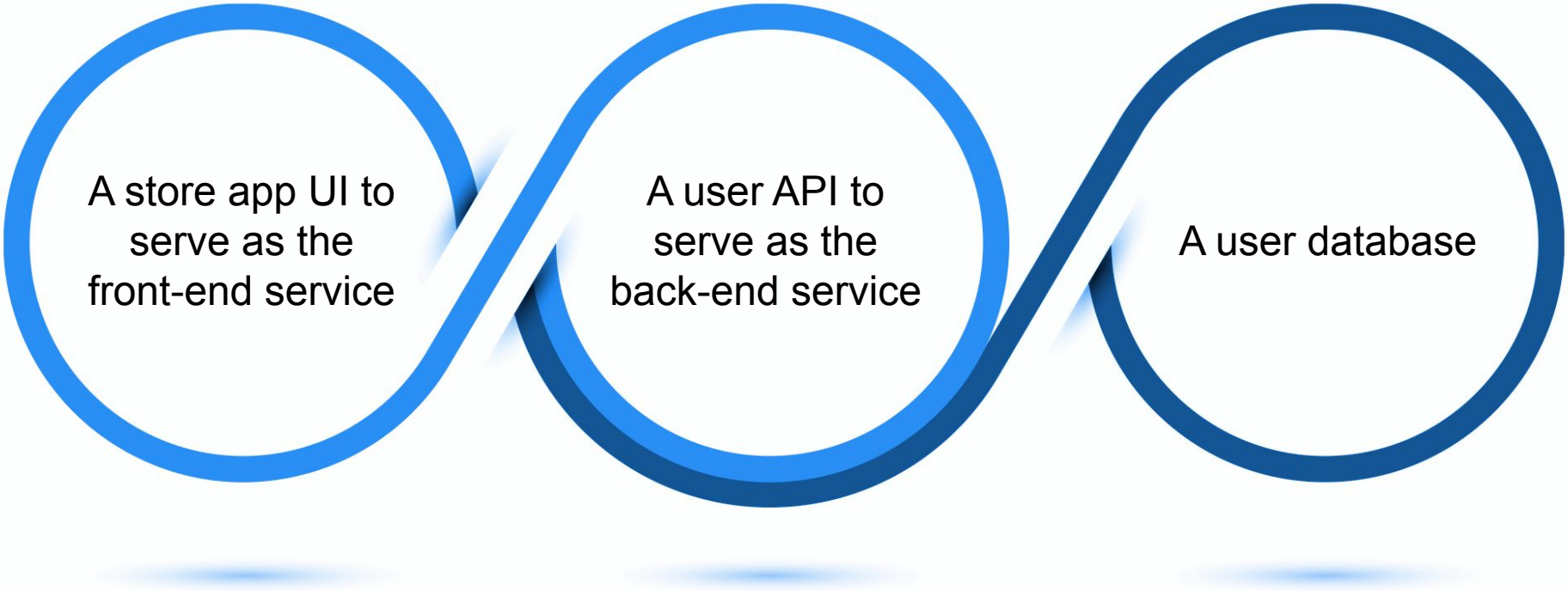


Microservice



Diagramming Microservices

If any component of the monolith is compromised, the entire application is compromised. Therefore, it is separated into the following components:



Diagramming Microservices

To make these changes on a diagram, we:

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

02 Add communication between each part of the microservice.

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

04 Rename the rest of the component services based on the main function they provide.

Activity: Monolith to Microservice

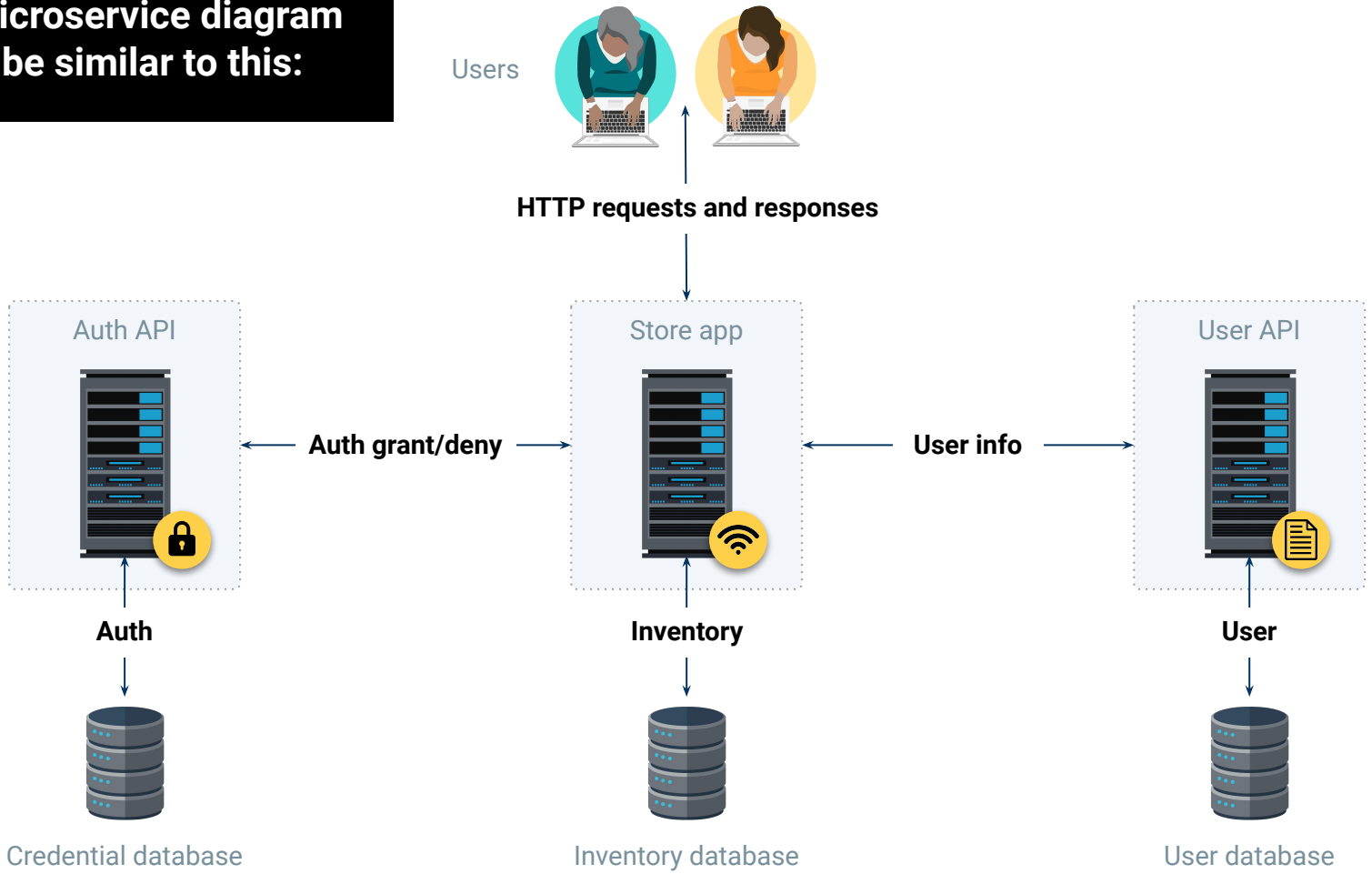
In this activity, you will use draw.io to illustrate a microservice version of a monolith web store application.

Suggested Time:

15 Minutes

Time's Up! Let's Review.

Your microservice diagram should be similar to this:



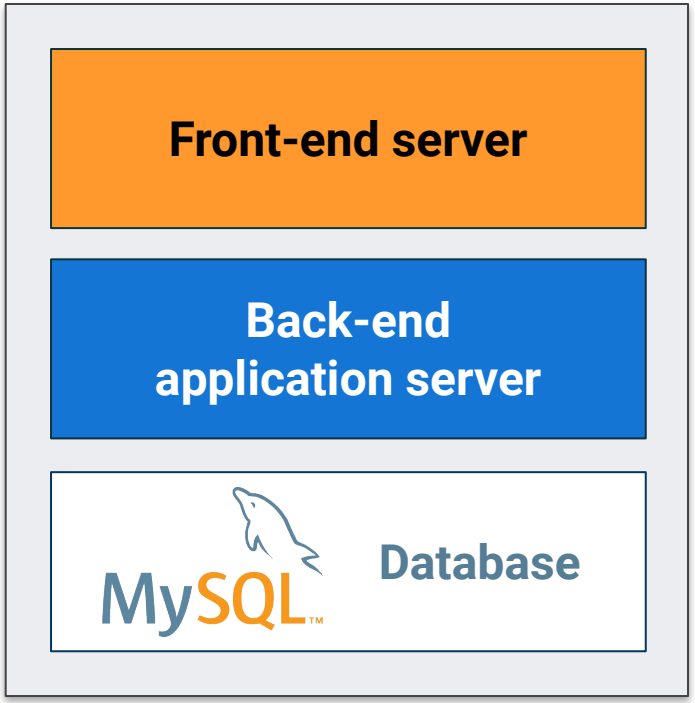
Questions?

Web Application Architecture

Web Application Architecture

One of the first steps in creating microservices is separating the components into their own individual machines so that each one runs 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 handling HTTP requests and responses.



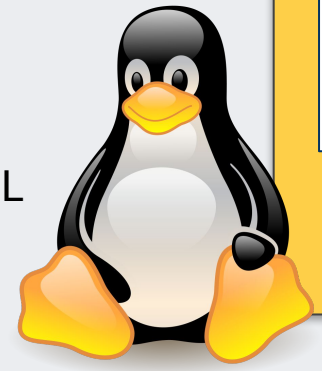
Web Application Architecture

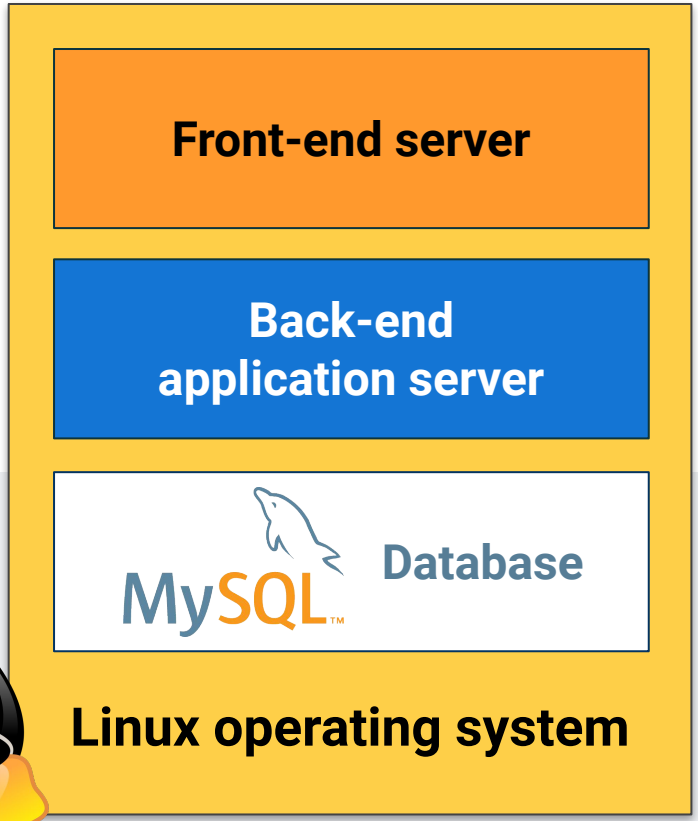
Each individual machine requires:

- 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 runs Linux as its operating system and has a MySQL database installed on it.

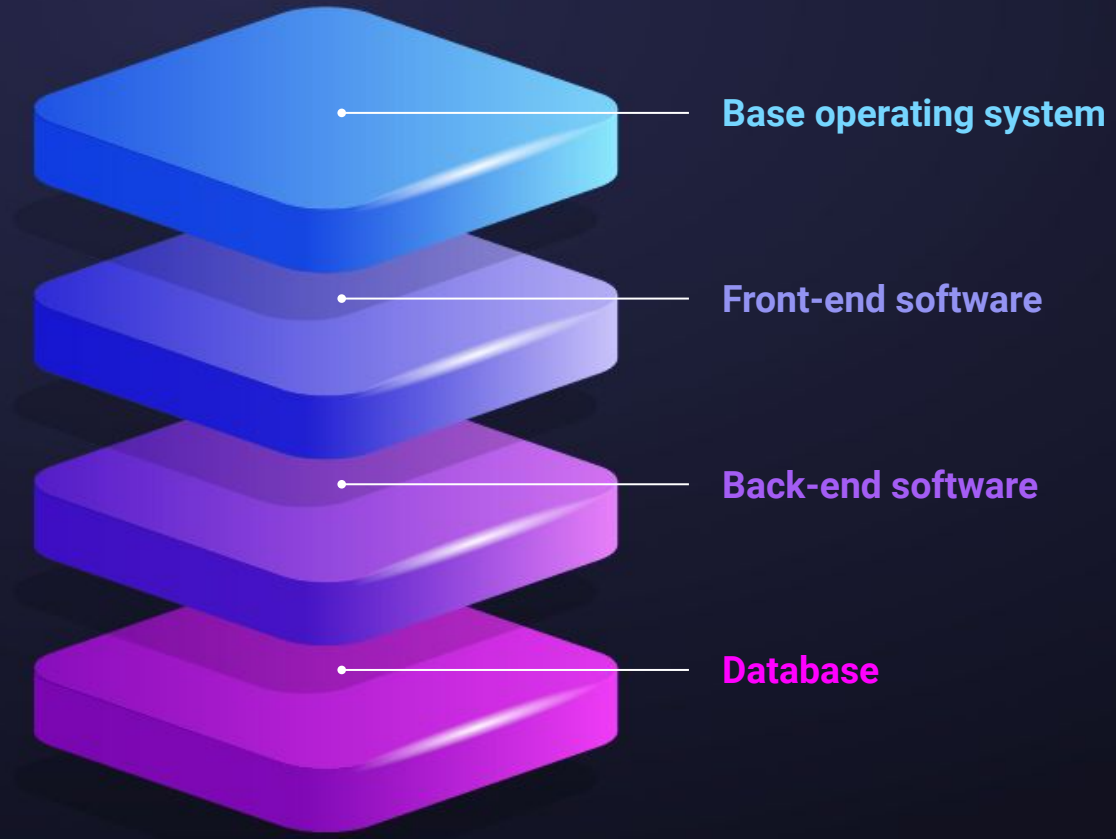




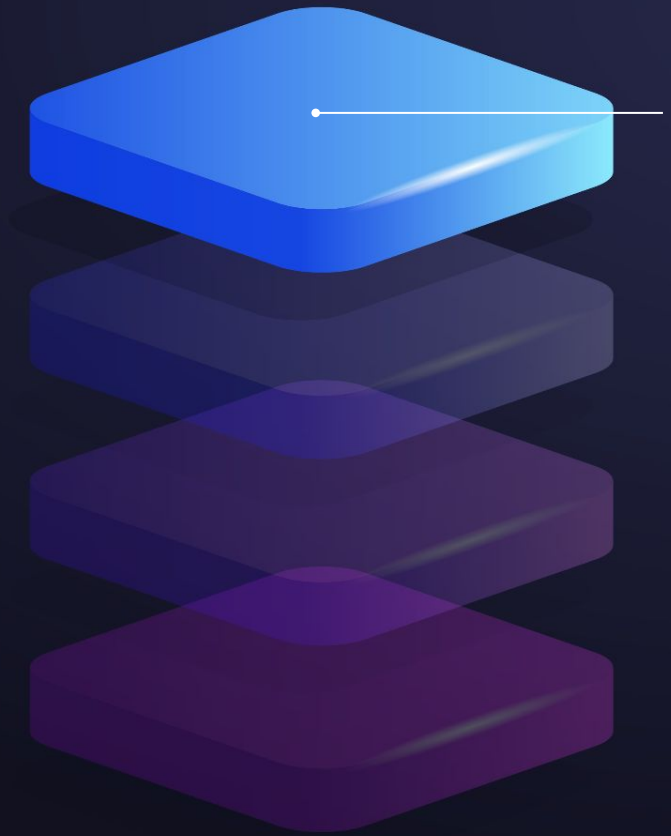
Introducing Stacks

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 database is called a **web stack**.



Introducing Stacks



Base operating system

Introducing Stacks

Front-end software

APACHE

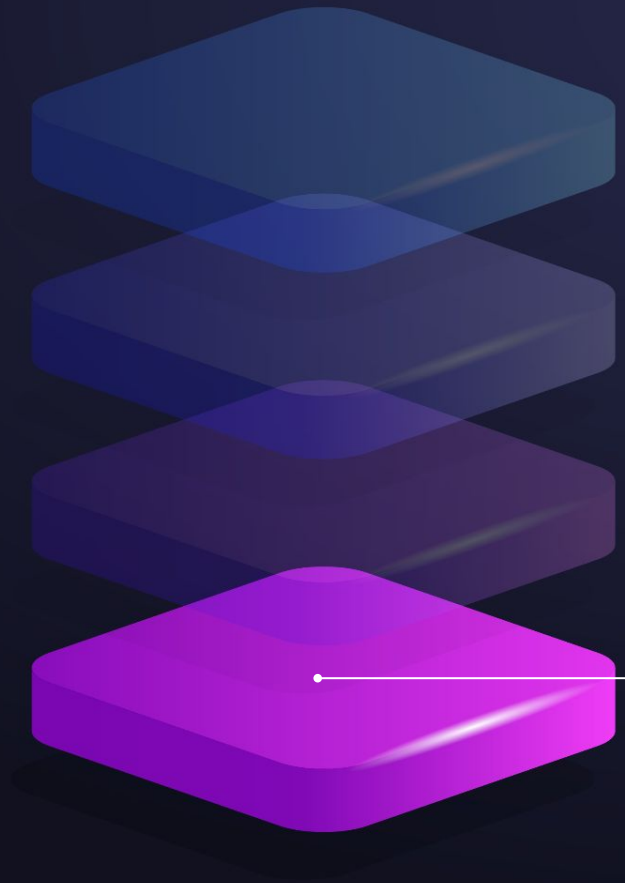
Microsoft IIS

NGINX

Introducing Stacks

Back-end software

Introducing Stacks



MySQL

Microsoft SQL Server

MariaDB

CouchDB

PostgreSQL

mongoDB

Database

LAMP Stack

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

Linux as the base operating system

Apache to handle the front-end HTTP requests & responses

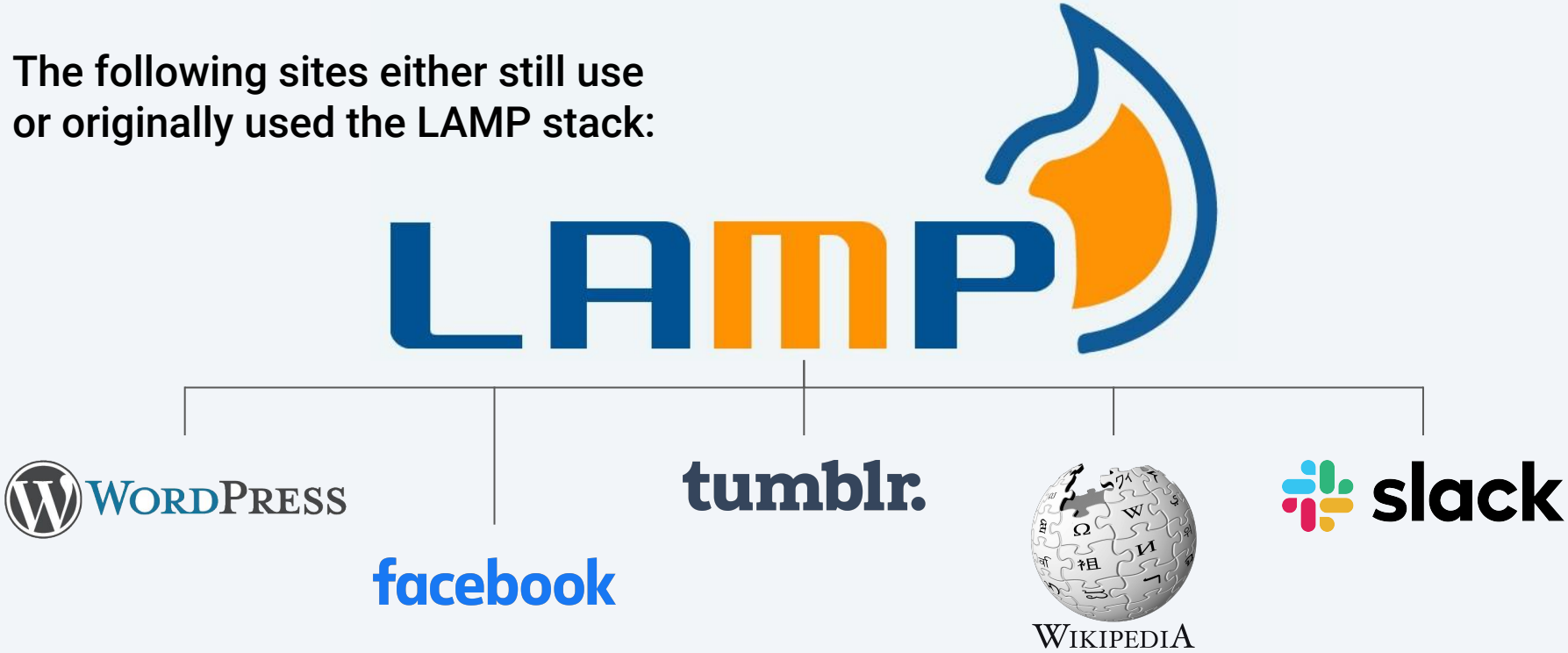
MySQL for storing data

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:

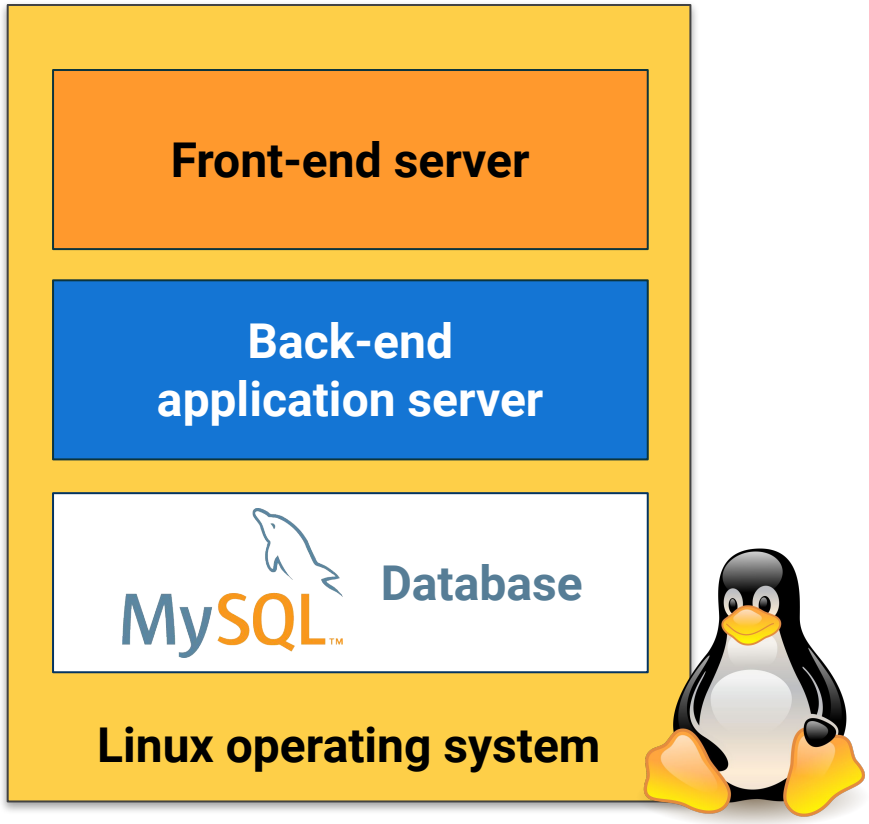


More Stacks

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 base operating systems are used for web applications:

Common Linux distros such as:

Windows Server 2016 and higher

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.
The front-end server is the part of a web application that renders the visual components of a web application.

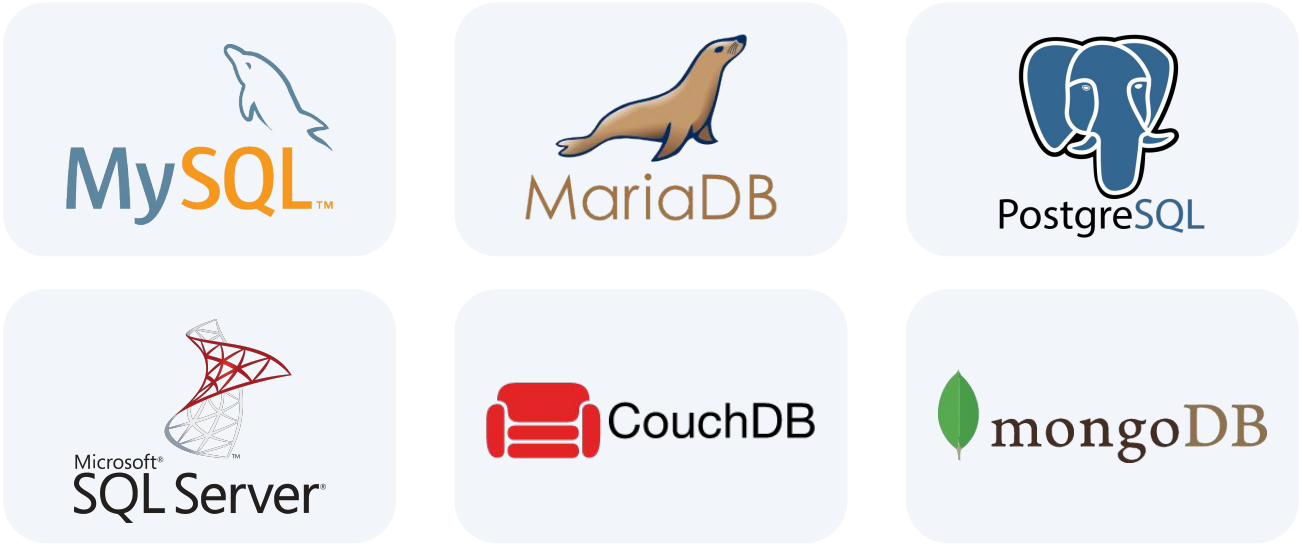
Primary front-end web design languages:

Specific examples of front-end servers:

Common Web Architecture: Database Server

Data for the application is housed in the database server. Remember that there could be multiple databases, some for storing sessions and cookies and some for storing business logic like a company's employee directory.

Specific examples of databases:



Common Web Architecture: Back-end Server

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

These servers run fully-featured scripting and programming languages.

Specific examples of back-end frameworks and languages:

Honorable Mention: The WIMP Stack

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

Windows operating system

Microsoft's **I**IS web server in place of Apache/Nginx

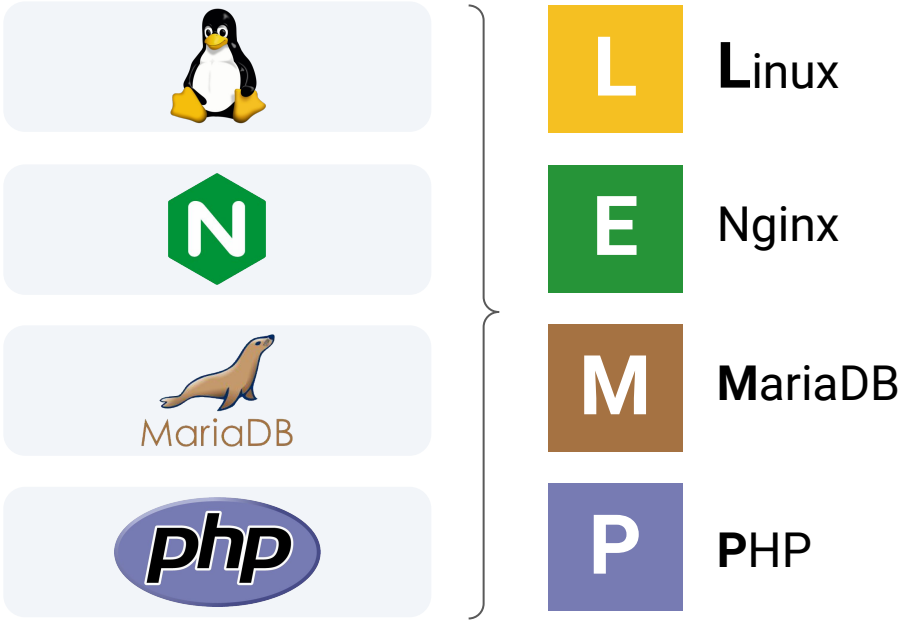
Database such as **M**icrosoft SQL Server

Back-end language such as **P**owerShell

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

Introducing the LEMP Stack

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



LEMP Stack

L

Linux

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

E

Nginx

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

M

MariaDB

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

P

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've 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 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.

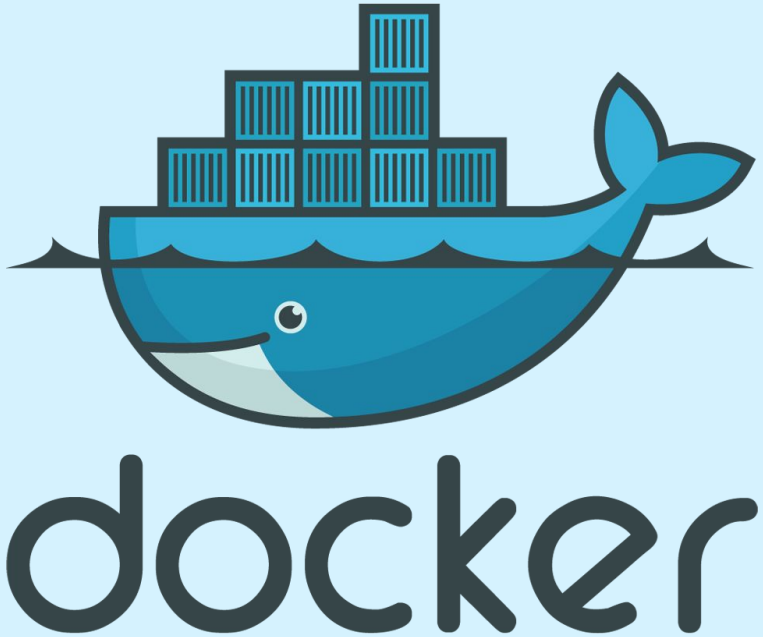
Full-sized VMs or physical machines for each service require more resources and are more expensive.

Each service needs its own lightweight virtual environment.

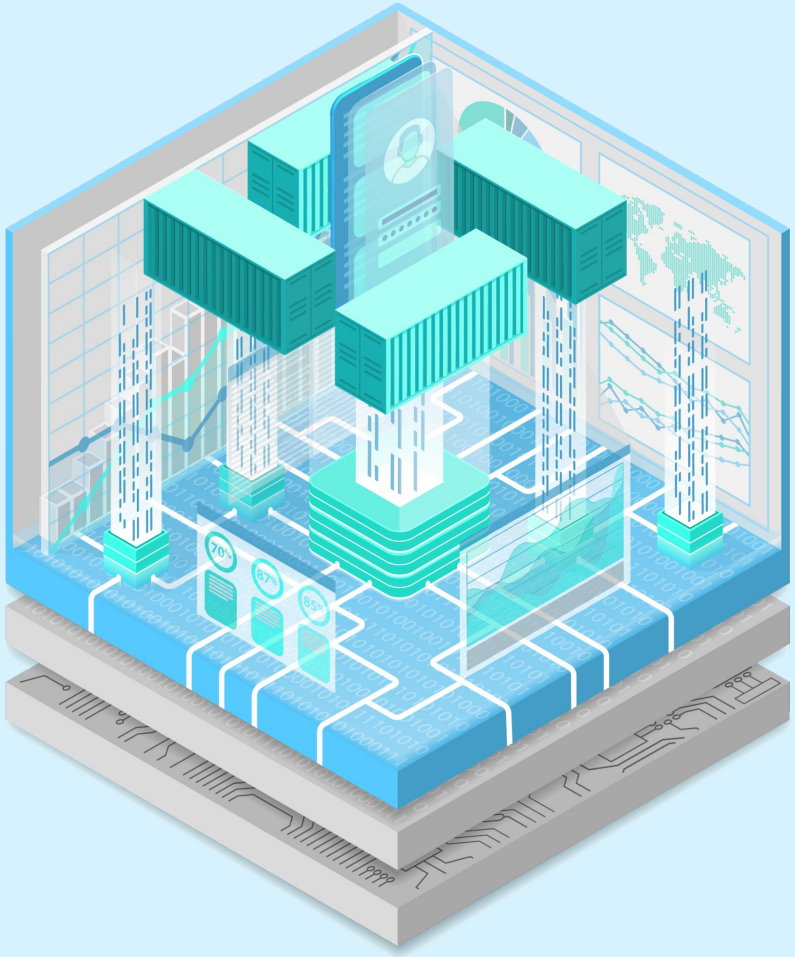
Can you think of any technologies that provide a lightweight, isolated environment?

Microservice Infrastructure with Containers

- 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 refer 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:

Declare a base operating system for the microservice to run on.

Copy the microservice's source code to the container.

Set a command that launches the microservice.

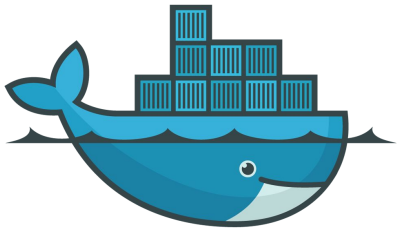
Dockerfile

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

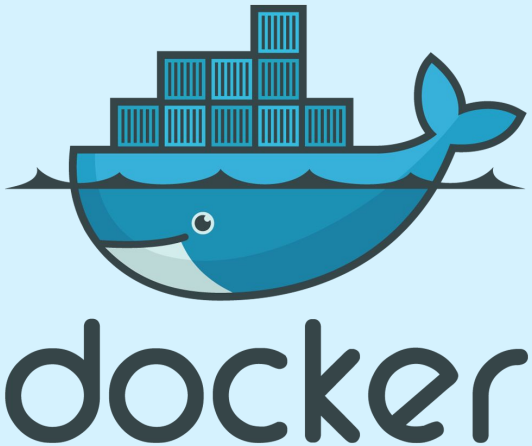
```
###################################
# Employee Directory Application
###################################
# Required Base OS
FROM ubuntu:14.04
  ...[truncated]
```

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

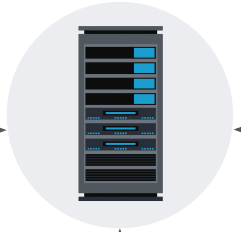
This is similar to an Ansible configuration file, which contains everything needed to configure a host with the command `ansible-playbook your-playbook.yml`.



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
```

Database

Dockerfile

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

Deploy and Test a Container Set

Deploying at Scale

Deploying containers can be tedious, especially if a business requires hundreds or thousands of microservices. 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)

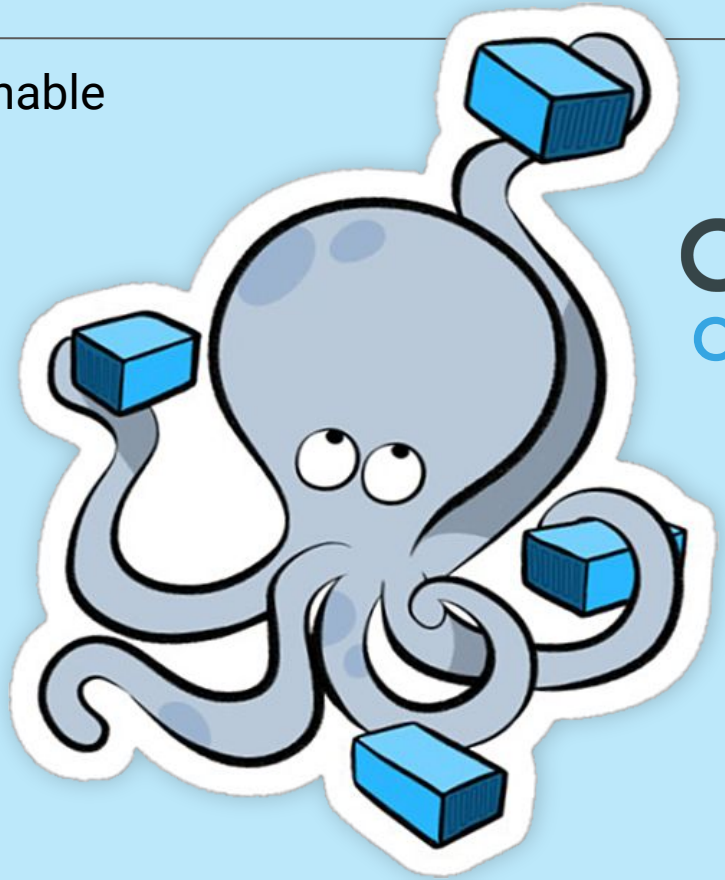
Set up a network for our containers so that they can communicate with each other

Preconfigure our containers, such as declaring the credentials for our web application's database

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

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 `docker-compose up`.

03 Use our browser to verify that our front-end user interface (UI) services are deployed properly.

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

Docker Compose YAML File

Docker Compose uses YAML to define the containers for a deployment, their networking configuration, and where you will copy files from your host machine into the 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…]
```

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:
```

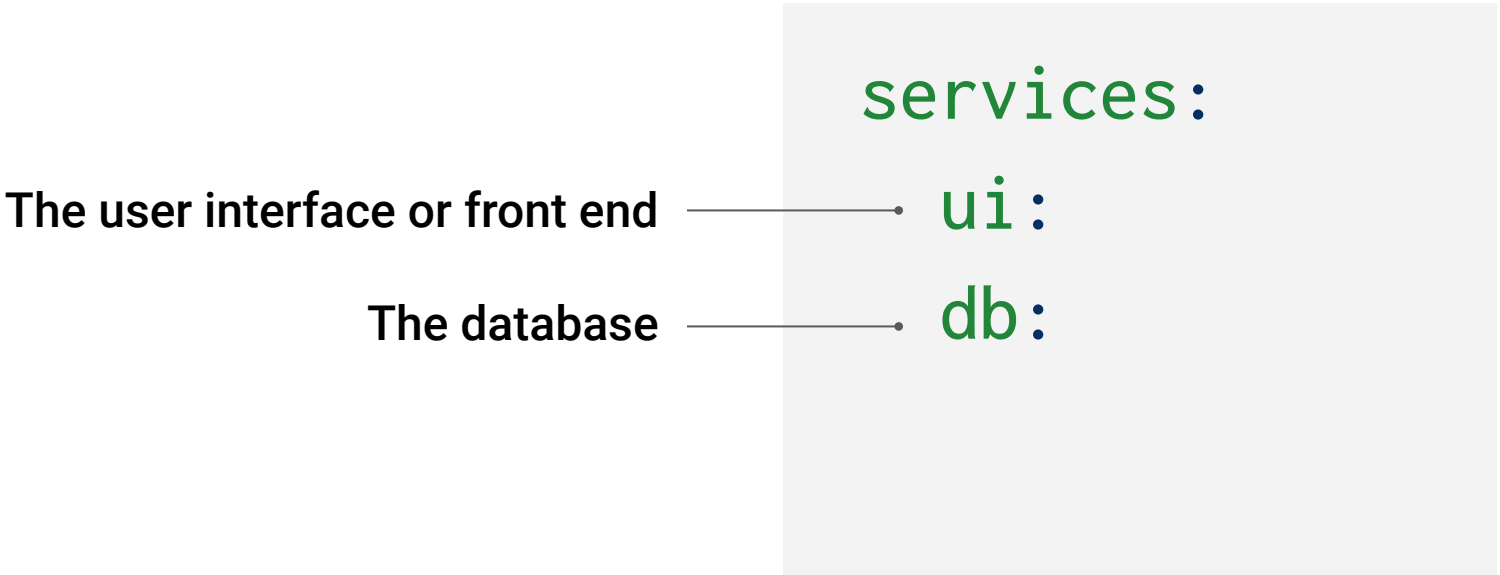
This is a
simple YAML file.

Let’s break it down.

YAML Breakdown

We'll break down the file by the **services** it installs.

In this YAML file, the `ui` and `db` are our services.



Docker Compose YAML File

`image:` 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.

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

We can find declared database credentials in the YAML file under the environment subheading.

`image:` The container image and version we'll use (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)

```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
```

Activity: Deploy and Test a Container Set

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

Suggested Time:

15 Minutes


Break

Time’s Up! Let’s Review.

Questions?

Introduction to Databases


Recap

Recap

Let’s recap everything we’ve learned so far today:

How to separate a monolith into microservices

How to describe web application architecture

Defining specific web application architectures

How to deploy and test that a container set is running and functional

Now that we have easily deployable microservice-based web applications...

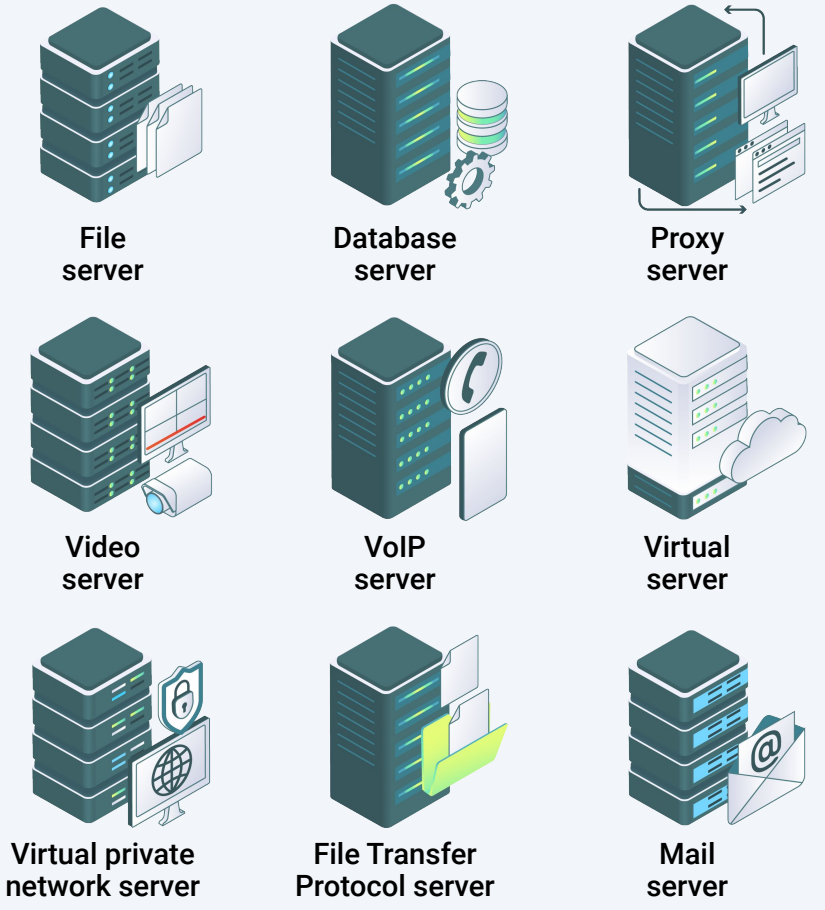
We'll cover one of the most critical components of a web application: the database.

The Importance of Databases

Databases contain the persistent data that flows through the web application.

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

- The usernames, telephone numbers, and email addresses of customers
- The session and cookie IDs of site visitors
- Personally identifiable information, 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 examine the threats and vulnerabilities of databases more closely later in the course.

Database Overview

Databases are used to store large amounts of data.

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

Database Overview

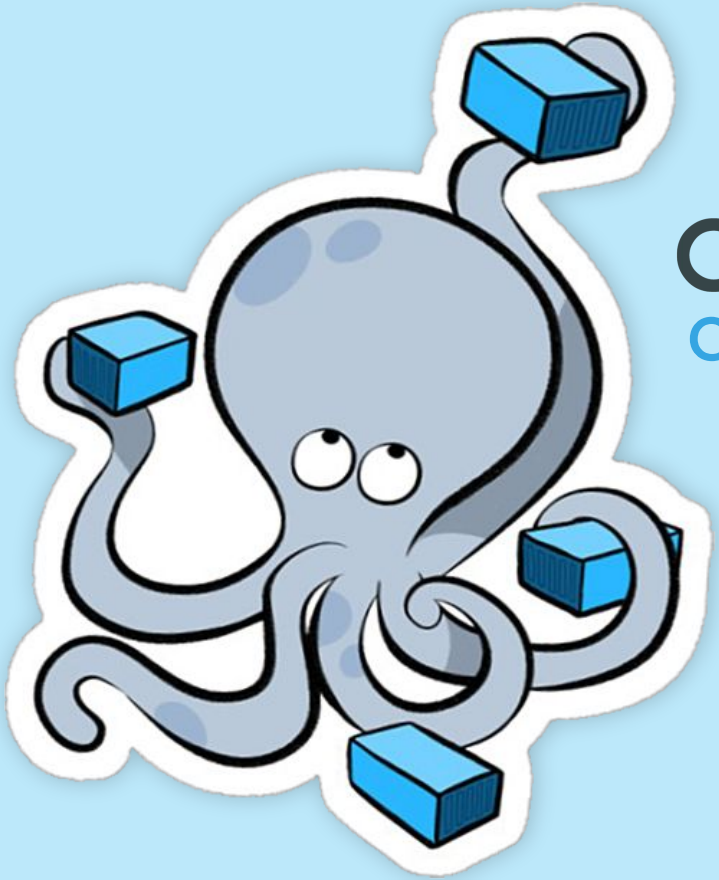
Databases are also typically kept separate from the other components of a web application.

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.

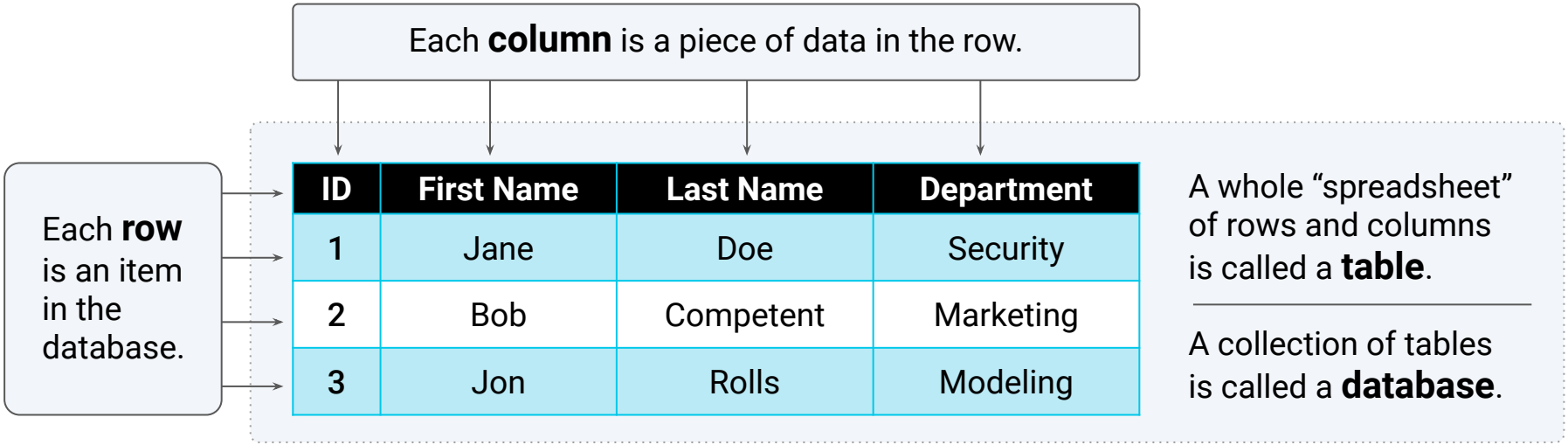


docker
compose

SQL Databases

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

SQL databases organize data like a spreadsheet.



Interacting with Data

Once you've created a table, there are four main ways to interact with the data it contains. Together, these are as known as **CRUD** operations.

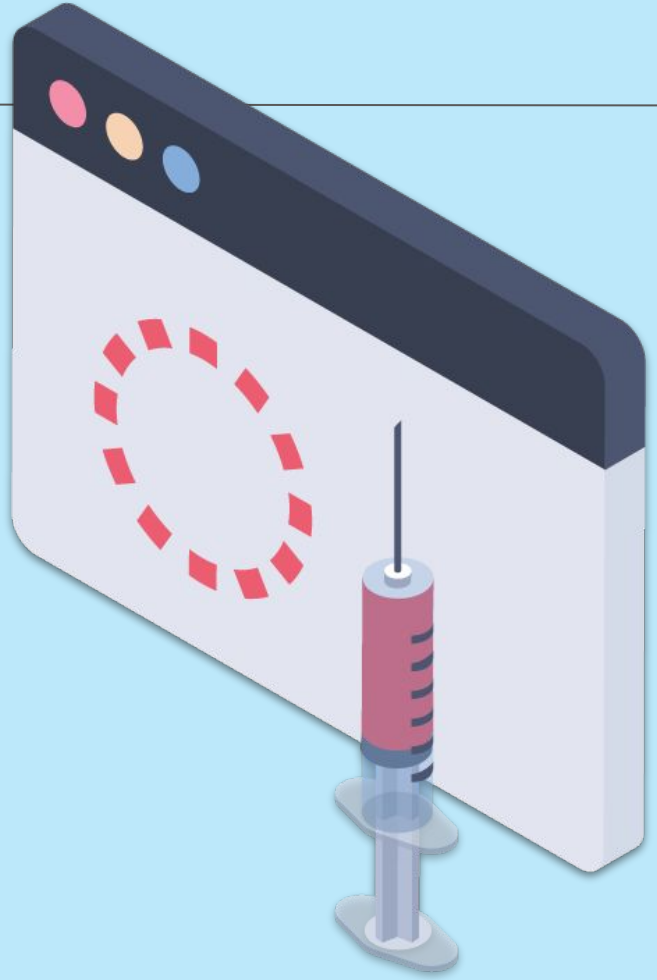
Create	**Add new entries to a database.**	When a user makes a new account in a web app, they are creating a new database entry with their user info.
Read	**View entries in a database.**	When a web application verifies your login credentials, it is reading the saved credentials it has in the database.
Update	**Modify existing data in a database.**	Whenever you need to reset your password in a web application, the password entry (more specifically, the hashed password) is updated with the new password.
Delete	**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 will be deleted from the database.

CRUD

CRUD operations are also known as **queries**.

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

We'll come back to the the SQL syntax of these operations to set the stage for **SQL injections** later in the course.



SQL injections are a type of web application attack intended to manipulate database 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 the following 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 in to the database.

04 Use a 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.

Questions?

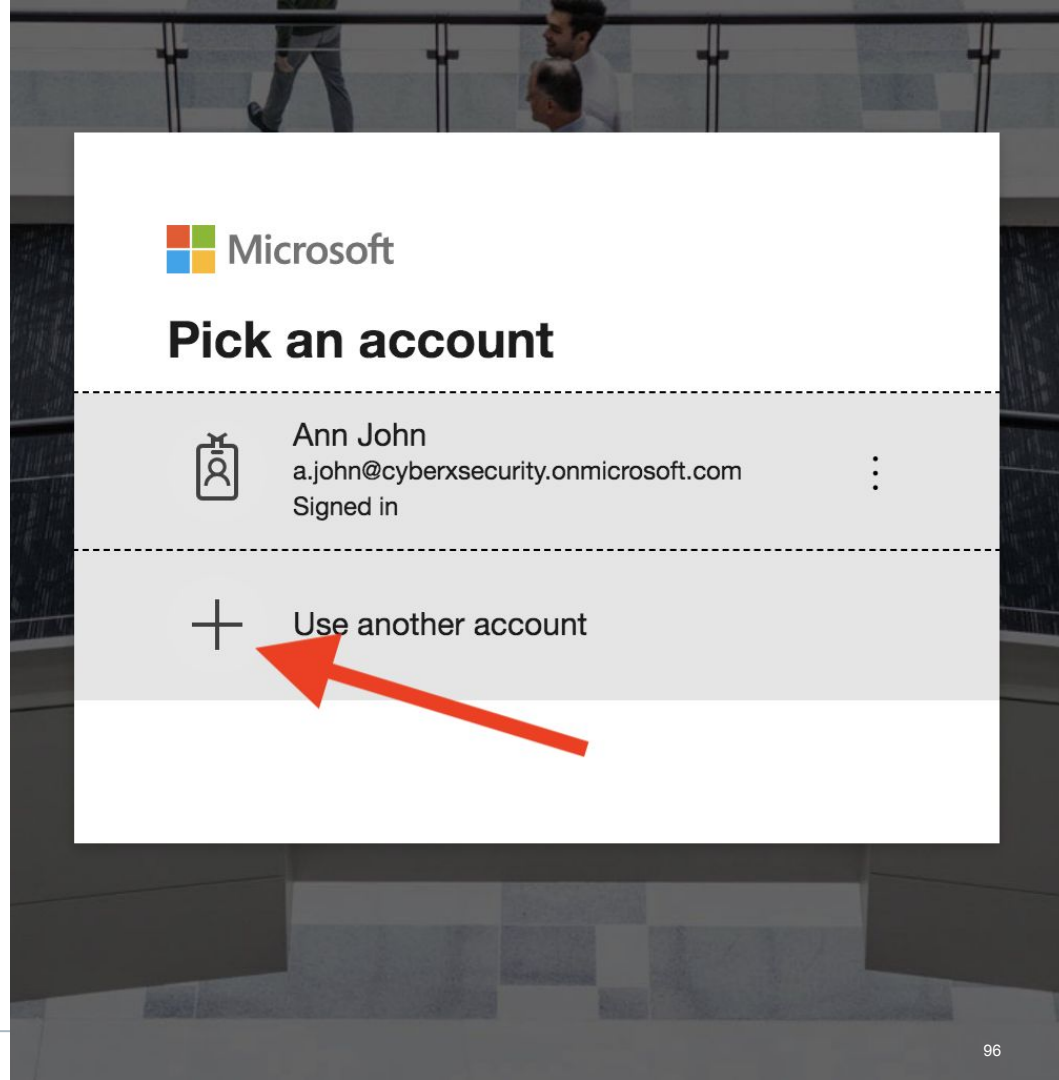
Personal Azure Account

For the upcoming Cloud Security and Project Week modules, you will need an **individual Azure account**.

You will not use the cyberxsecurity account for these modules.

Refer to the following guide:

[Personal Azure Account Setup Guide]




The End