**What are Microservices?**

Microservices are a way of building applications where each part (or "service") of the application operates independently but communicates with the other parts. Each service is responsible for a specific function, like user authentication, payments, or managing game scores, and they all work together to run the overall application. This allows you to update or change individual parts without affecting the whole system.

**How Docker Enables Microservices:**

1. **Isolation**: With Docker, each microservice can run in its own **container**, completely isolated from the others. This means you can have different services (like the login system, game service, or leaderboard service) running separately, with their own software, libraries, and configurations. Even if one service fails, it won’t take down the whole system.
2. **Portability**: Docker makes each microservice self-contained and **portable**. You can move these containers between development, testing, and production environments without worrying about the different setups. This ensures that the microservice works the same, no matter where it’s running.
3. **Scalability**: Microservices often need to scale independently. For example, if the game service is getting a lot of traffic, you can spin up more containers for that service only, without affecting the other parts of BucStop. Docker allows you to easily scale these microservices up and down as needed.
4. **Simplified Management**: Docker makes it easy to manage multiple microservices through tools like **Docker Compose** or **Kubernetes**. You can define how different services should run and interact, simplifying deployment and orchestration.
5. **Consistency**: With Docker, each microservice has the same environment every time it runs. This removes "it works on my machine" issues because the container includes everything needed to run the service, regardless of where it's deployed.

**Example with BucStop:**

In BucStop, you could break down the website into microservices:

* **Authentication Microservice**: Handles user login and school email verification.
* **Game Microservice**: Runs individual games like Snake, Pong, and Tetris.
* **Leaderboard Microservice**: Tracks and displays high scores for each game.

Each of these services would run in its own Docker container, allowing them to function independently but communicate with each other. For instance, the **game microservice** can send scores to the **leaderboard microservice**. If you need to update the leaderboard, you only have to update that container without touching the game or authentication services.

**Summary:**

Docker makes microservices possible by providing:

* **Isolation** for each service.
* **Portability** across environments.
* **Scalability** to handle different loads.
* **Consistency** to ensure reliability.
* **Ease of management** for deploying and controlling multiple services.

Docker allows you to break down a large, complex application like BucStop into smaller, manageable, and scalable parts—making microservices an effective approach.

**Explanation:**

**BucStop** is a website where students can play games like Snake, Pong, and Tetris. To make sure these games run smoothly, they each need specific software, settings, and environments—like how a cake needs the right ingredients and a specific oven temperature. Docker helps by packing everything needed for these games and the website into **containers**.

Think of containers as “boxes” that hold everything each game needs to run, like software, instructions, and settings. By using Docker, it doesn’t matter where BucStop is hosted (on your computer or a server online), everything will work exactly the same every time because Docker keeps all the requirements inside those boxes.

**Important Vocabulary:**

1. **Docker**: A tool that helps package applications and their settings into **containers** so they run consistently, no matter where they're hosted.
2. **Containers**: Think of containers like sealed boxes that hold everything a specific game or the website needs to run. Each container is self-sufficient, meaning it has its own resources, software, and settings, and doesn’t need help from the outside system.
3. **Docker Images**: These are like blueprints for containers. A Docker image contains everything needed to create a container, such as the code and all the software required to run it. It’s a template that Docker uses to build and run containers.
4. **Dockerfile**: This is a set of instructions Docker uses to build an image. It tells Docker what software and settings to include in the container. For example, it may say, "Start with a base game, then add the specific instructions to run Snake."
5. **Docker Compose**: A tool used when BucStop has multiple parts (like different games or services) that need to work together. Docker Compose lets you define and run all these parts (containers) at the same time in a simple way, by using a file that describes everything the website needs.
6. **Image Registry (like Docker Hub)**: Think of this as a public library of Docker images. You can store your BucStop game containers here, or download ready-made ones that other people have shared.
7. **Portability**: This term refers to Docker’s ability to ensure that the games on BucStop run exactly the same way, no matter which computer or server they're being used on. Docker's containers make it easy to move the application anywhere without having to adjust for different environments.

**Example Using BucStop:**

* BucStop’s website is like a party where you have different games (Snake, Pong, Tetris) set up.
* Each game needs its own setup (controllers, rules, scoreboards), and with Docker, you can pack all of this for each game into its own container (its own box).
* When it's time for the party (launching the website), you just open each box (container) and the games run perfectly without needing extra setup or worrying about compatibility issues on different computers or servers.

By using Docker, BucStop ensures that everything works smoothly across different environments, saving you from technical headaches!

where or when it’s played.

Helpful Links

<https://docs.docker.com/get-started/docker-overview/>

<https://docs.docker.com/get-started/docker-overview/>

**Steps to Deploy:**

Create an Ubuntu EC2 Instance on AWS and Generate a .pem Key

1. Log into the AWS Management Console
   1. Go to [AWS Management Console](https://aws.amazon.com/console/).
   2. If you don’t have an account, you’ll need to sign up for one.
2. Navigate to EC2
   1. In the search bar at the top, type "EC2" and select it from the results.
   2. This will take you to the EC2 dashboard.
3. Click "Launch Instance"
   1. On the EC2 dashboard, click the orange Launch instance button to create a new EC2 instance.
4. Configure the EC2 Instance
   1. Name your instance (optional): Give your instance a name for easy reference.
   2. Select an Amazon Machine Image (AMI):
      1. In the "Amazon Machine Image (AMI)" section, select Ubuntu Server 22.04 LTS (or any other version of Ubuntu you prefer).
   3. Choose an Instance Type:
      1. Choose an instance type based on your project needs. For most projects, the t2.micro instance is sufficient and falls under the free tier.
   4. Key Pair (Login) - Create a New Key Pair:
      1. In the "Key pair (login)" section, click Create a new key pair.
      2. Name the key pair (e.g., bucstop-key).
      3. Ensure the .pem file format is selected.
      4. Choose **RSA** as the key pair type when creating the .pem file.
      5. Click Create key pair and download the .pem file. Store this .pem file in a secure location (e.g., C:/Users/YourUsername/AWSKeys).
5. Configure Network Settings - Allow HTTP Traffic
   1. Under Network settings, make sure you check the box for Allow HTTP traffic from the internet. This is important for allowing access to your EC2 instance over HTTP (port 80).
   2. If you need HTTPS traffic as well, select Allow HTTPS traffic from the internet (optional).
   3. You should **leave "Allow SSH traffic" selected**. SSH traffic (port 22) is necessary for you to securely connect to your EC2 instance from your local machine using the .pem key.
6. Launch the EC2 Instance
   1. Leave other settings as default unless you have specific requirements.
   2. Scroll to the bottom and click **Launch instance**.
7. View Instance Status
   1. Once the instance is launched, you will be redirected to the **Instances** page where you can see your instance initializing.
   2. Wait for the instance to reach the **Running** state.
8. Retrieve the Public DNS/IPv4 Address
   1. Once your instance is running, click on the instance ID to view the instance details.
   2. Copy the **Public IPv4 address** or **Public DNS** as you’ll need it to connect to the instance.
9. Status check will say initializing. The EC2 instance goes through a short **initialization process** after it's launched. Once it's finished, the status will change to **2/2 checks passed** and the instance state will show as **Running**. You can refresh the page occasionally to see when it finishes initializing. Once it's in the **Running** state, you'll be ready to move on to the next step.

Connect to the EC2 Instance via SSH:

1. Open Your Local Directory with the .pem Key
   1. Open **Git Bash** or **Command Prompt** on your local machine.
   2. Navigate to the directory where you saved the .pem key file (e.g., C:/Users/YourUsername/AWSKeys). Use the following command to navigate to the directory: “cd /c/Users/YourUsername/AWSKeys”.
2. Copy the SSH Command from AWS
   1. Go to your AWS EC2 dashboard.
   2. Select your instance, then click **Connect**.
   3. Go to the **SSH** tab.
   4. AWS will provide an SSH command like this: “ssh -i "YourKey.pem" ubuntu@Your-EC2-Public-DNS”. Replace "YourKey.pem" with the name of your .pem key file, and use the **Public DNS** or **IPv4 address** of your instance.
3. Run the SSH Command in Git Bash
   1. In Git Bash, paste and run the SSH command provided by AWS: “ssh -i "YourKey.pem" ubuntu@Your-EC2-Public-DNS”.
   2. Example: “ssh -i "bucstop-key.pem" [ubuntu@ec2-12-34-56-78.compute-1.amazonaws.com](mailto:ubuntu@ec2-12-34-56-78.compute-1.amazonaws.com)”
4. Verify Connection: You should now connect to your EC2 instance, and it will prompt you with a message like: “Are you sure you want to continue connecting (yes/no)?”.
   1. Type yes and hit Enter.

Create deployadmin User and Grant Sudo Privileges:

1. **Create the deployadmin User**: Run the following command to create a new user named deployadmin: “sudo adduser deployadmin”
   1. It will ask you to enter some information like full name and password. You can set a simple password and leave most of the fields blank (just press Enter to skip them).
2. **Grant deployadmin Sudo Privileges**: Run this command to add deployadmin to the sudo group, giving them administrative privileges: “sudo usermod -aG sudo deployadmin”
3. **Switch to the deployadmin User**: Now switch to the deployadmin user: “sudo su – deployadmin”

Create a scripts Folder:

1. **Create the scripts Folder**: Run the following command to create a folder called scripts in the home directory: “mkdir ~/scripts”
2. **Navigate to the scripts Folder**: Use the cd command to go into the scripts folder: “cd ~/scripts”

Add the Script Files from GitHub:

1. **Open Git Bash on Your Local Machine (not logged into the EC2 instance, you may need more than 1 gitbash window open)**: Open a terminal (Git Bash or any terminal) on your local machine where the script files are stored.
2. **Navigate to the Directory Where the Script Files Are Located**:
   1. Use cd to go to the directory where your scripts folder is stored locally (e.g., /c/Users/jasmi/source/repos/CodeJunkies): “cd /c/Users/jasmi/source/repos/CodeJunkies”
3. **Use scp to Transfer the scripts Folder to the EC2 Instance**:
   1. Run the following scp command to transfer the files: scp -i "C:/Users/jasmi/AWSKeys/bucstop-key.pem" -r scripts/ [deployadmin@ec2-18-117-115-92.us-east-2.compute.amazonaws.com:/home/deployadmin/scripts/](mailto:deployadmin@ec2-18-117-115-92.us-east-2.compute.amazonaws.com:/home/deployadmin/scripts/)
   2. This will securely copy the scripts folder from your local machine to the scripts folder on your EC2 instance.
4. If you get the error that Permission denied (publickey): indicates that there’s an issue with authenticating the deployadmin user using the .pem file. Here’s how we can resolve it:
   1. **Ensure the Key File is Correct**: Make sure that the bucstop-key.pem you’re using is the correct private key for the EC2 instance. It should match the key you used to create the instance and is associated with the deployadmin user.
   2. **Check if the Public Key for deployadmin is Set Up**: Since we’ve added the deployadmin user manually, we need to ensure the SSH public key is set up for that user.
      1. **Log in as the ubuntu user** (which should work, since it was created automatically): “ssh -i "C:/Users/jasmi/AWSKeys/bucstop-key.pem" [ubuntu@ec2-18-117-115-92.us-east-2.compute.amazonaws.com](mailto:ubuntu@ec2-18-117-115-92.us-east-2.compute.amazonaws.com)”
      2. **Switch to the deployadmin user**: “sudo su – deployadmin”
      3. **Create the .ssh directory for deployadmin** (if it doesn’t exist) and set correct permissions: “mkdir -p ~/.ssh chmod 700 ~/.ssh”
      4. **Set Up the authorized\_keys File**: If you haven't already, copy the public key from the .pem file to the authorized\_keys file for deployadmin.
         1. **Generate the public key** from your .pem file on your **local machine** (in a Git Bash window on your local machine): “ssh-keygen -y -f C:/Users/jasmi/AWSKeys/bucstop-key.pem”
         2. **Copy the public key** output from the above command. DO NOT SHARE THIS KEY VALUE WITH CHAT GPT
         3. **On your EC2 instance (logged in as deployadmin)**, open the authorized\_keys file: nano ~/.ssh/authorized\_keys
         4. **Paste the public key** you generated from your .pem file.
         5. **Save and exit** the file (press CTRL + X, then Y, then Enter).
      5. **Set the Correct Permissions**: “chmod 600 ~/.ssh/authorized\_keys”
   3. **Try the scp Command Again**: Now that the public key is correctly set up for deployadmin, try the scp command again from your local machine.

Set the Correct Permissions for the .pem file on your machine:

1. Open Git Bash on your local machine.
2. Navigate to the folder where your .pem key is stored: “cd /c/Users/jasmi/AWSKeys”
3. **Set the correct permissions for the .pem file:** Run the following chmod command to set the correct permissions for your .pem key: “chmod 400 bucstop-key.pem”. This ensures that only you, the owner, can read the file, which is required by AWS to allow secure connections.

Now, let's set up permissions on your EC2 instance for the deployadmin user:

1. **Log in to your EC2 instance** (if you're not already logged in): “ssh -i "C:/Users/jasmi/AWSKeys/bucstop-key.pem" [ubuntu@ec2-18-117-115-92.us-east-2.compute.amazonaws.com](mailto:ubuntu@ec2-18-117-115-92.us-east-2.compute.amazonaws.com)”
2. **Switch to the deployadmin user**: “sudo su – deployadmin”
3. Create the .ssh directory (if it doesn't already exist): “mkdir -p ~/.ssh

chmod 700 ~/.ssh”

1. **Check if the authorized\_keys file exists**: If the authorized\_keys file exists, we’ll ensure it has the correct permissions. If it doesn’t exist, you will create it and paste your public key there.
   1. Ensure You’re Logged in as deployadmin. If you’re not already logged in as the deployadmin user, run the following commands to log into your EC2 instance and switch to the deployadmin user: “ssh -i "C:/Users/jasmi/AWSKeys/bucstop-key.pem" ubuntu@ec2-18-117-115-92.us-east-2.compute.amazonaws.com
   2. sudo su – deployadmin”
   3. **Check if the authorized\_keys File Exists.** Run this command: “ls -l ~/.ssh/authorized\_keys”. If the file exists, the output will show details about the file (like permissions, owner, etc.). If the file doesn’t exist, you’ll get an error like: “ls: cannot access '/home/deployadmin/.ssh/authorized\_keys': No such file or directory”
   4. If the authorized\_keys file doesn’t exist, you need to create it.
      1. **Generate the public key** from your .pem file on your local machine. In a new Git Bash window on your local machine, run: “ssh-keygen -y -f C:/Users/jasmi/AWSKeys/bucstop-key.pem”
      2. **Copy the public key** from the output of this command.
      3. **Create the authorized\_keys file** on your EC2 instance: “nano ~/.ssh/authorized\_keys”
      4. **Paste the public key** into the file by right-clicking in the terminal.
      5. Save the file. In nano, press CTRL + X, then press Y, and hit **Enter** to save and exit.
   5. Set the Correct Permissions for authorized\_keys. Whether the file existed or you just created it, set the correct permissions for the authorized\_keys file: “chmod 600 ~/.ssh/authorized\_keys”. This ensures that only the deployadmin user can access the file.
2. **Set permissions for the authorized\_keys file**: Run the following command to set the correct permissions: “chmod 600 ~/.ssh/authorized\_keys”. This makes sure that only the deployadmin user can read or write to this file.
3. Once you've set the correct permissions, retry the scp command from your local machine to transfer the scripts folder.