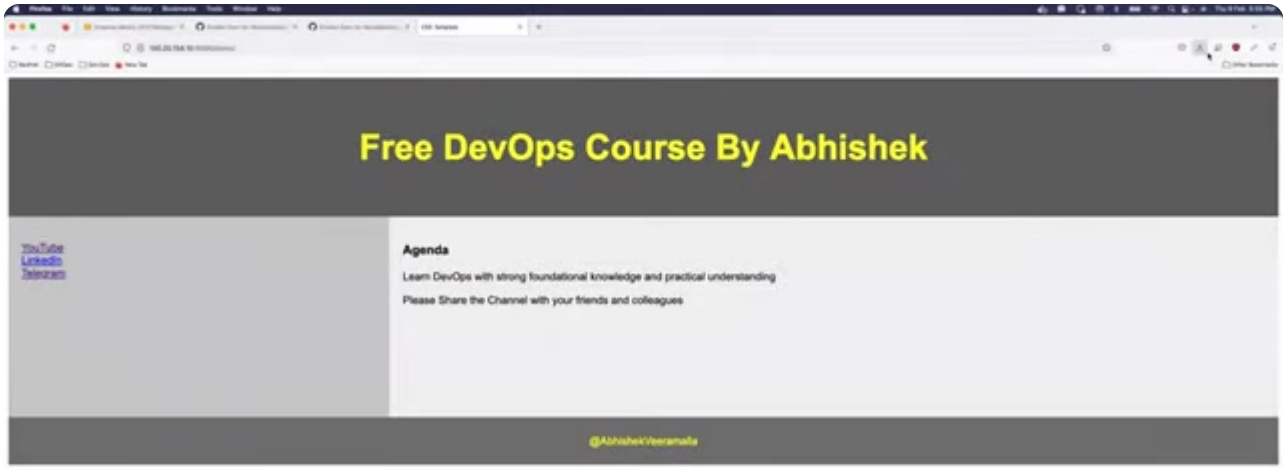
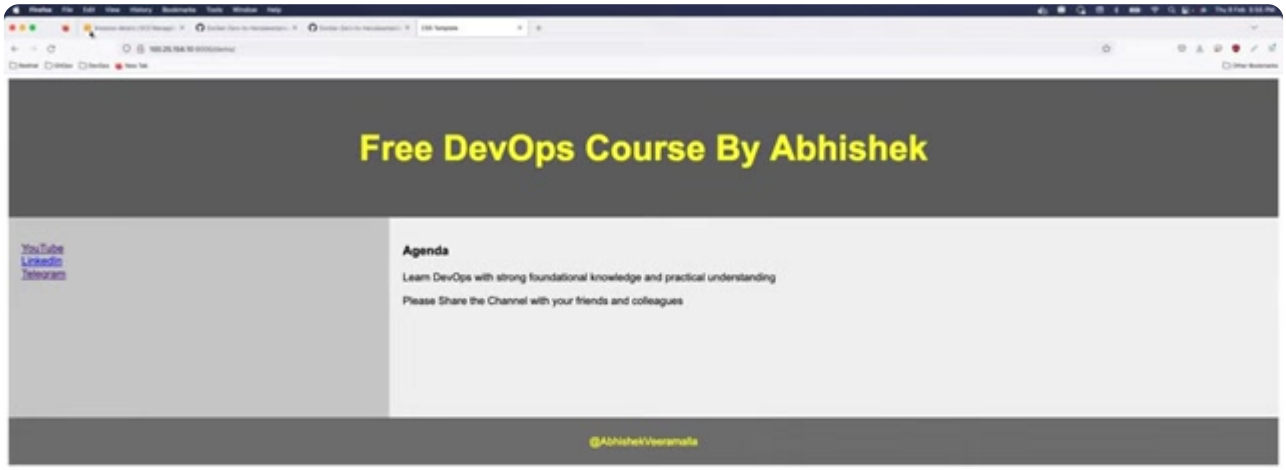
Introduction [00:00:02]



Hello everyone, my name is Abhishek, and welcome back to my channel. Today is Day 25 of our complete DevOps course, and in this class, we will deploy our first Django web application as a Docker container.

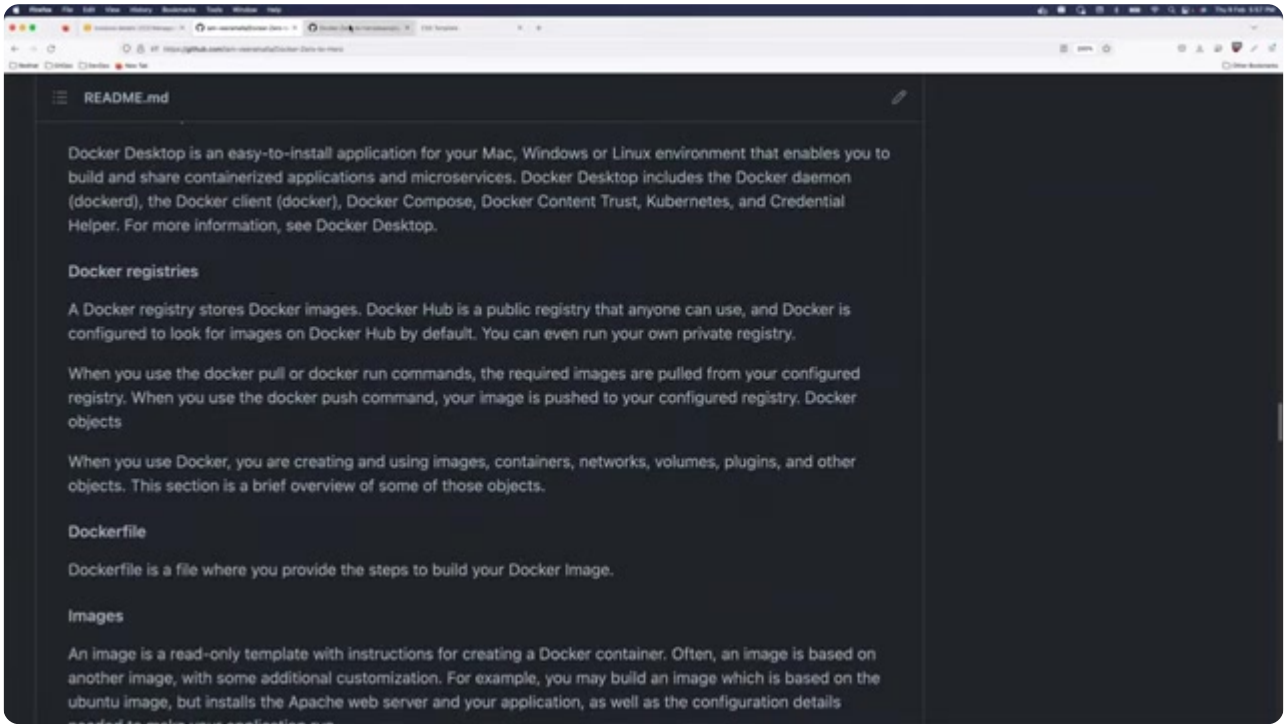
Recap of Previous Classes (Day 23 & 24) [00:00:18]



In the previous classes (Day 23 and Day 24), we covered the fundamentals of Docker. If you go to the `iam-veeramalla/Docker-Zero-to-Hero` GitHub repository, you'll find everything we discussed:

- The concept of containers and why they are lightweight compared to virtual machines.
- The file and folder structure within a container.
- The Docker Architecture, including the Client, Docker Daemon, and Registry.
- The Docker Lifecycle, from a Docker image to a Docker container.
- Essential Docker terminology like Daemon, Client, Desktop, Registries, Dockerfile, and Image.
- We installed Docker on an EC2 instance, managed permissions, and successfully ran a `docker run hello-world` application.

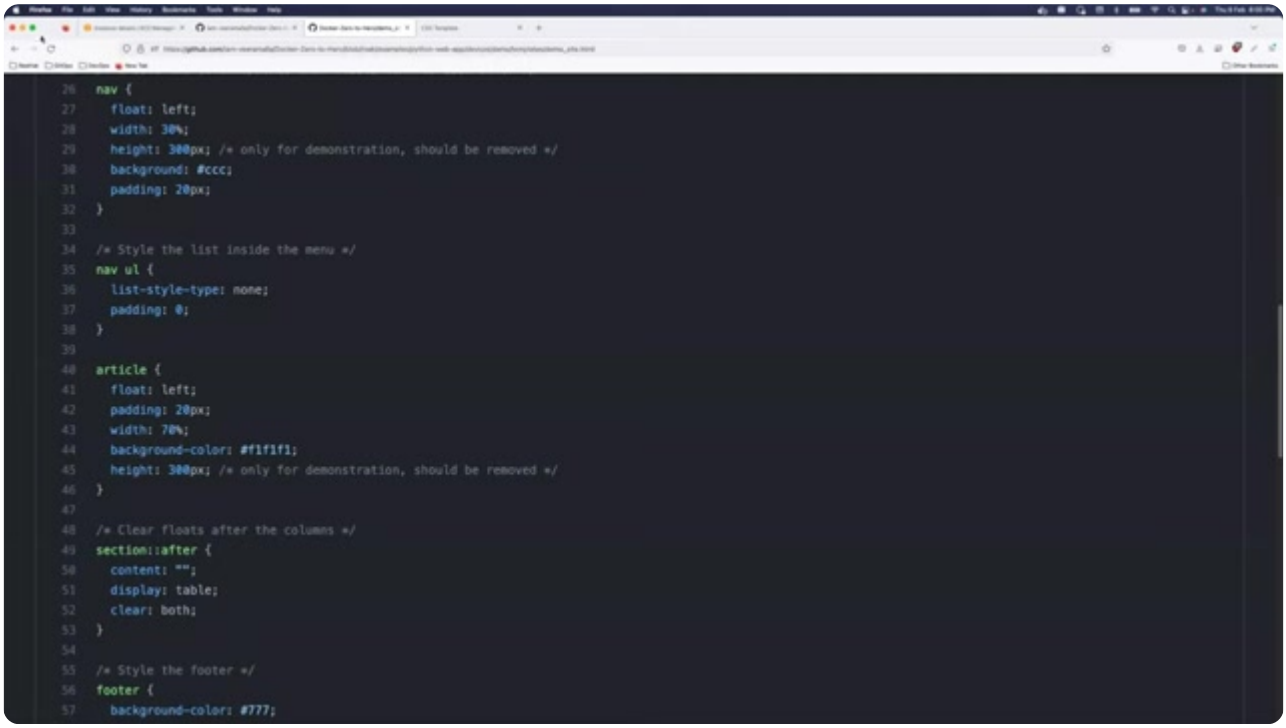
Introducing the Django Application & The DevOps Role [00:01:54]



Today, we'll work with a Django application that I've written. It's a very basic single-page application, but the process of containerizing it is the same whether it's a single page or has multiple flows.

A common question is, "Do I need to have programming experience as a DevOps engineer?" The answer is that you should have a basic idea of how applications function. You don't need to write the application from scratch, but you must understand its structure to containerize it effectively. For example, you should be able to look at this project and understand what the `requirements.txt` file is for and what the `devops` source code folder contains. This knowledge allows you to build the correct Docker image.

Django Application Workflow Explained [00:04:36]



```
nav {
  float: left;
  width: 30%;
  height: 300px; /* only for demonstration, should be removed */
  background: #ccc;
  padding: 20px;
}

/* Style the list inside the menu */
nav ul {
  list-style-type: none;
  padding: 0;
}

article {
  float: left;
  padding: 20px;
  width: 70%;
  background-color: #f1f1f1;
  height: 300px; /* only for demonstration, should be removed */
}

/* Clear floats after the columns */
section::after {
  content: "";
  display: table;
  clear: both;
}

/* Style the footer */
footer {
  background-color: #777;
```

Before we containerize the application, I'll take a few minutes to explain its workflow.

1. **Create a Project**: First, a Django project is created, which acts as a skeleton. This is done with the command `django-admin startproject <project_name>`. In our case, the project is named `devops`. This is similar to how `ansible-galaxy` creates a skeleton for Ansible roles.

2. **Project Configuration**: Inside the project, you have key files:

   - `settings.py`: Contains all the project-level configurations, such as allowed hosts, database settings, secret keys, and middleware.

   - `urls.py`: Manages URL routing. It maps URL paths to specific views.

3. **Create an Application**: Within a project, you create applications. This is done with `python manage.py startapp <app_name>`. Our application is named `demo`.

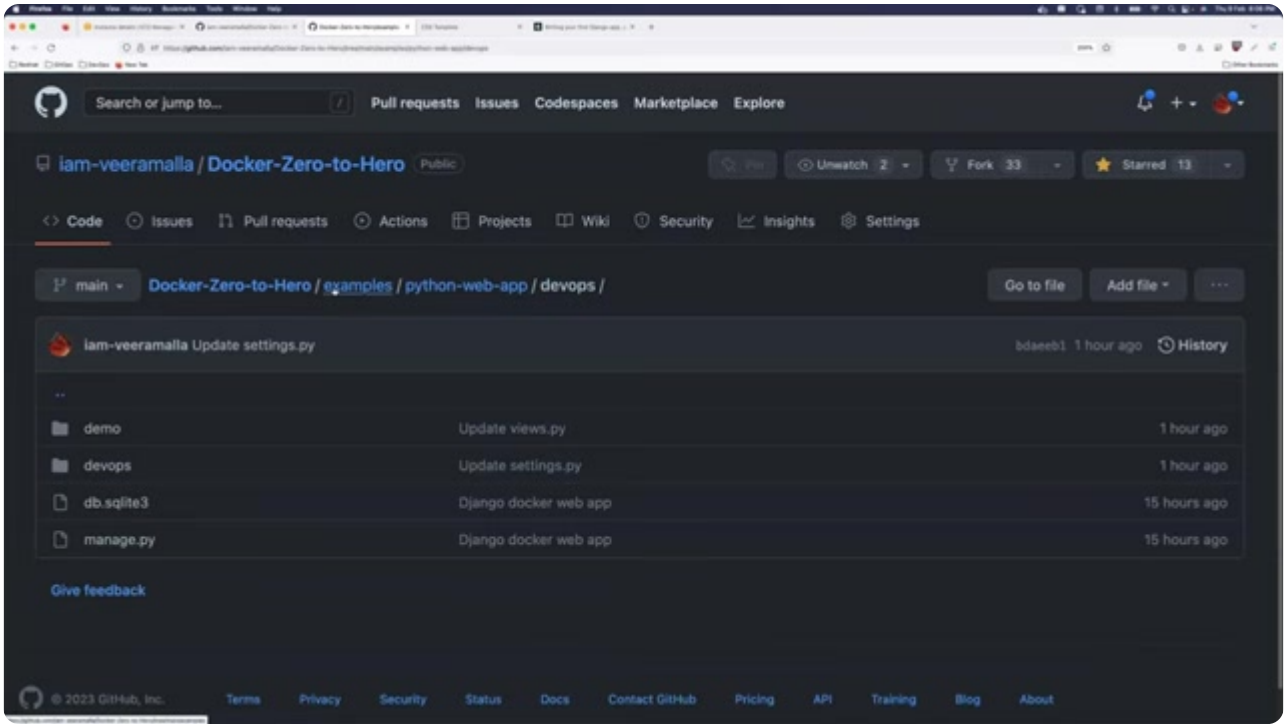
4. **Application Logic**: The `startapp` command creates more files for the application's logic.

   - `views.py`: This is where the main Python code for the application resides.

   - `templates`: This folder holds the HTML files that will be rendered.

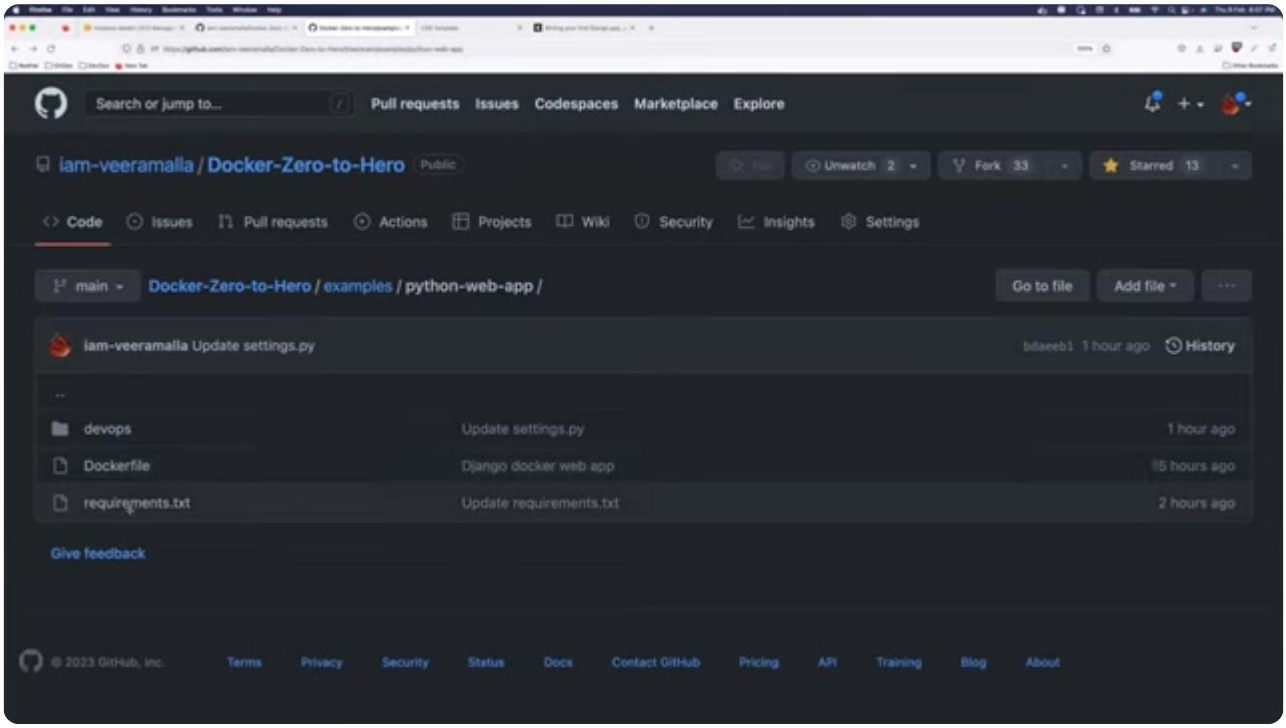
5. **Request Flow**: When a user accesses the application, `urls.py` directs the request. In our setup, a request to `/demo` is routed to the `index` function in `views.py`, which in turn renders the `demo_site.html` template.

The "Works on My Machine" Problem [00:11:26]



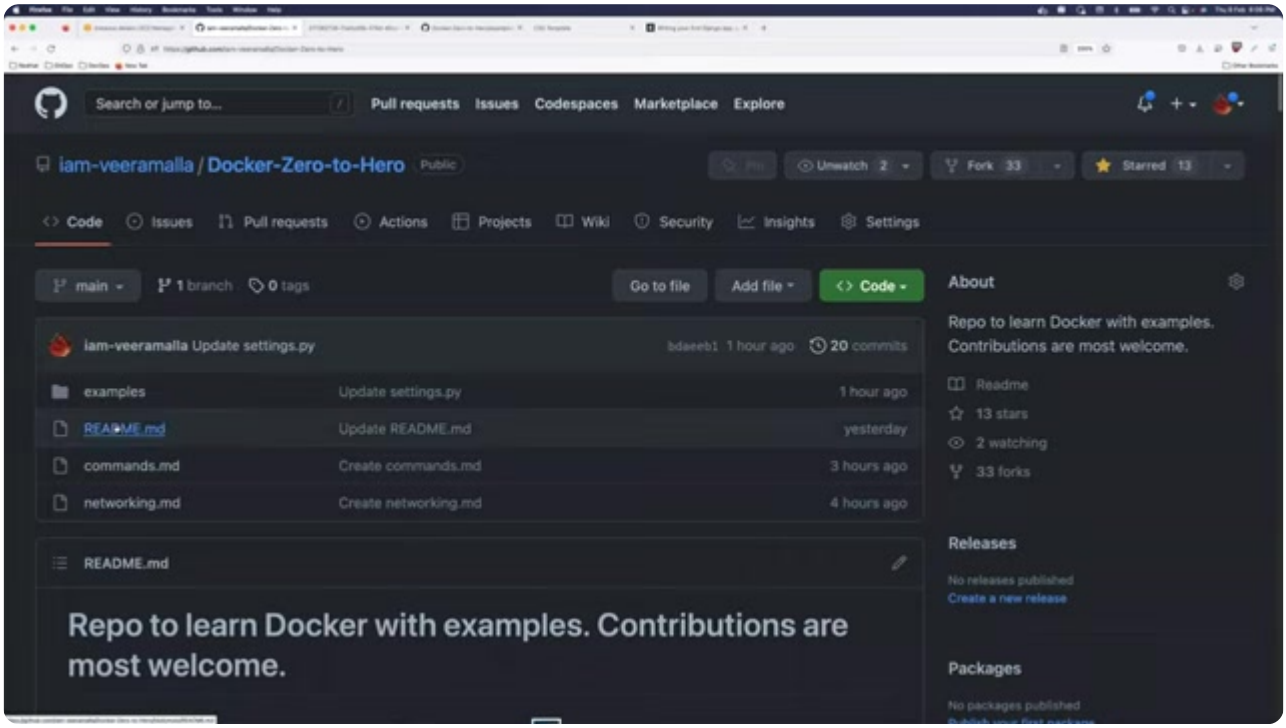
Before containers, deploying an application was challenging. A developer would finish their code, and the QA team would have to set up the entire environment on their own machines. They would clone the repository and install all the dependencies from `requirements.txt`. This process often failed because the QA engineer might be on a different operating system (Windows, macOS, or another Linux distribution) than the developer. This led to the classic "it works on my machine" problem.

How Docker Solves the Problem [00:12:27]



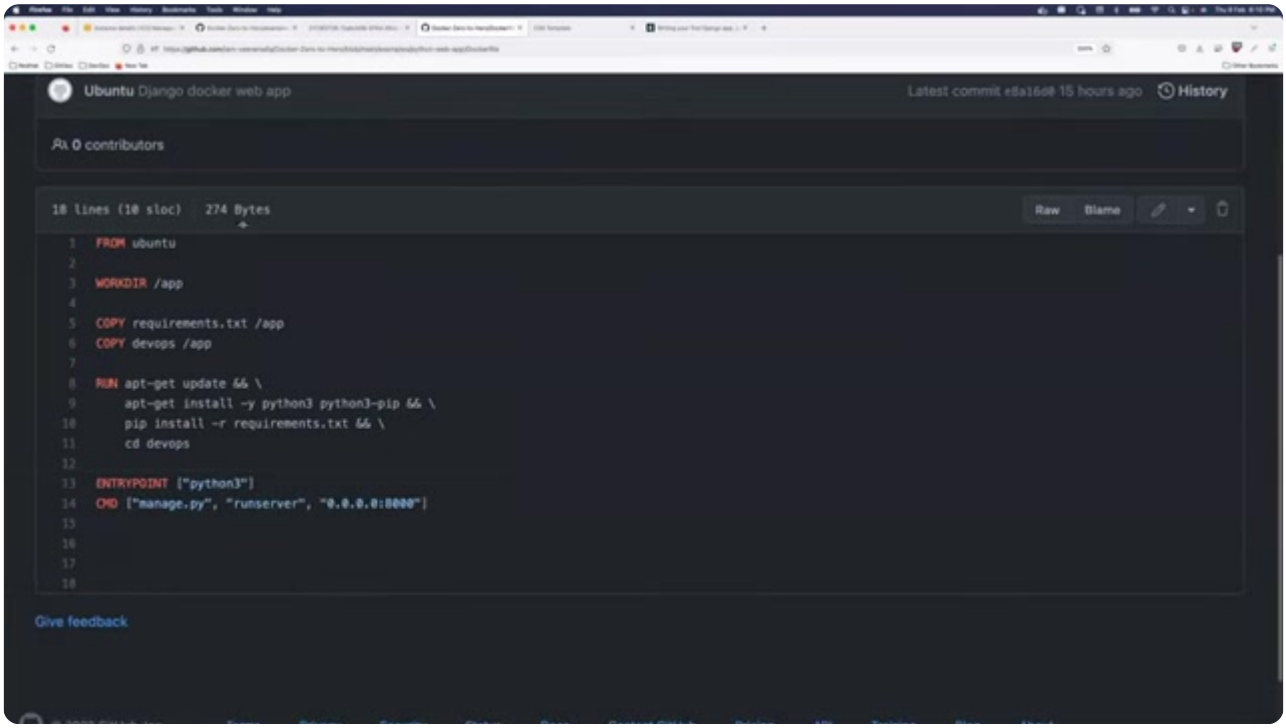
Docker containers solve this classic problem. A container bundles the application source code along with all of its dependencies—like the specific version of Python and required libraries—into a single, portable package. This container image will run the exact same way on any machine that has Docker installed, regardless of the host operating system. The container has its own minimal system dependencies and only uses the host OS kernel for system calls, ensuring a consistent environment everywhere.

The DevOps Engineer's Role in Containerization [00:14:01]



As a DevOps engineer, your task is to containerize this Django application. The first step is to analyze the application's needs and write a `Dockerfile`. I have already written one and included it in the repository.

Dockerfile Walkthrough [00:15:05]



Let's break down the `Dockerfile`:

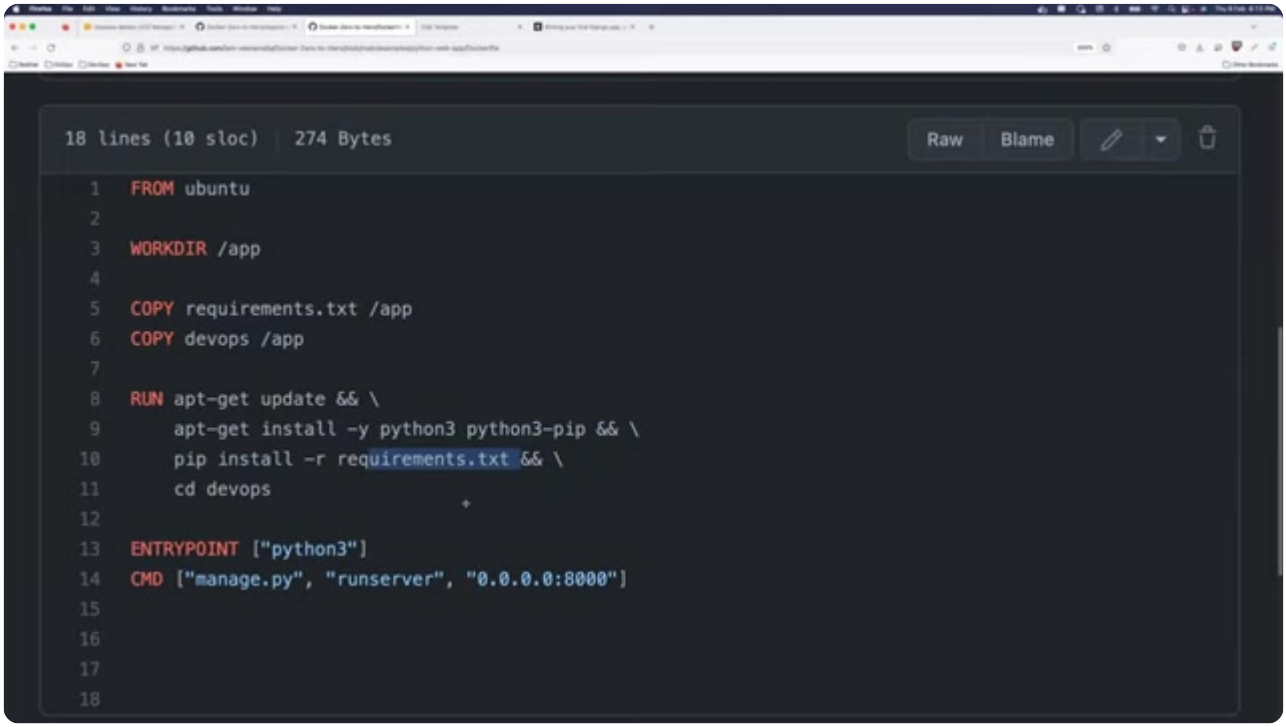
```
FROM ubuntu

WORKDIR /app

COPY requirements.txt /app

COPY devops /app

RUN apt-get update && \

    apt-get install -y python3 python3-pip && \

    pip install -r requirements.txt

ENTRYPOINT ["python3"]

CMD ["manage.py", "runserver", "0.0.0.0:8000"]
```

- `FROM ubuntu`: We start with a base Ubuntu image. You could also start with a `python` base image, which would simplify the installation steps.
- `WORKDIR /app`: This sets the working directory inside the container to `/app`. All subsequent commands will be run from this directory.
- `COPY requirements.txt /app` & `COPY devops /app`: We copy the `requirements.txt` file and

the `devops` source code directory into the container's `/app` directory.

- `RUN ...` : This command first updates the package list, then installs Python and pip. Finally, it uses pip to install the application's dependencies listed in `requirements.txt` .
- `ENTRYPOINT ["python3"]` : This sets the main executable for the container. The `ENTRYPOINT` is not meant to be overridden at runtime.
- `CMD ["manage.py", "runserver", "0.0.0.0:8000"]` : This provides the default arguments to the `ENTRYPOINT` . The `CMD` can be easily overridden by the user when they run the container.

ENTRYPOINT vs. CMD Explained [00:17:38]



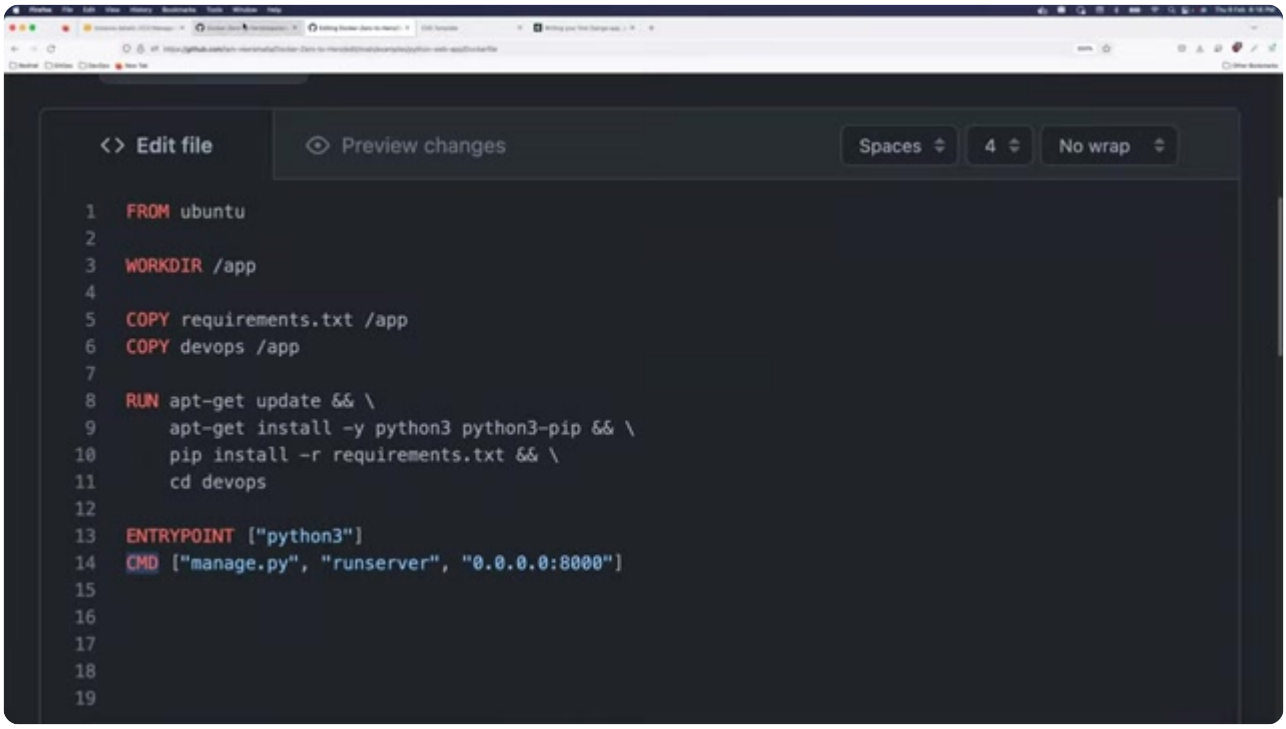
The difference between `ENTRYPOINT` and `CMD` is a common question.

- Both define the command that runs when the container starts.
- `ENTRYPOINT` specifies the main executable that should not be changed. In this case, it's `python3` .
- `CMD` provides default arguments that can be configured or overridden by the user. Here, it tells Python to run the Django development server.

This separation is a best practice. It allows a user to change the port or other `runserver` arguments

without having to change the core executable. If you put everything in `CMD`, a user could accidentally change the executable from `python3` to something else. By using `ENTRYPOINT`, you create a more stable and predictable container.

Live Demonstration: Building and Running the Container [00:21:33]



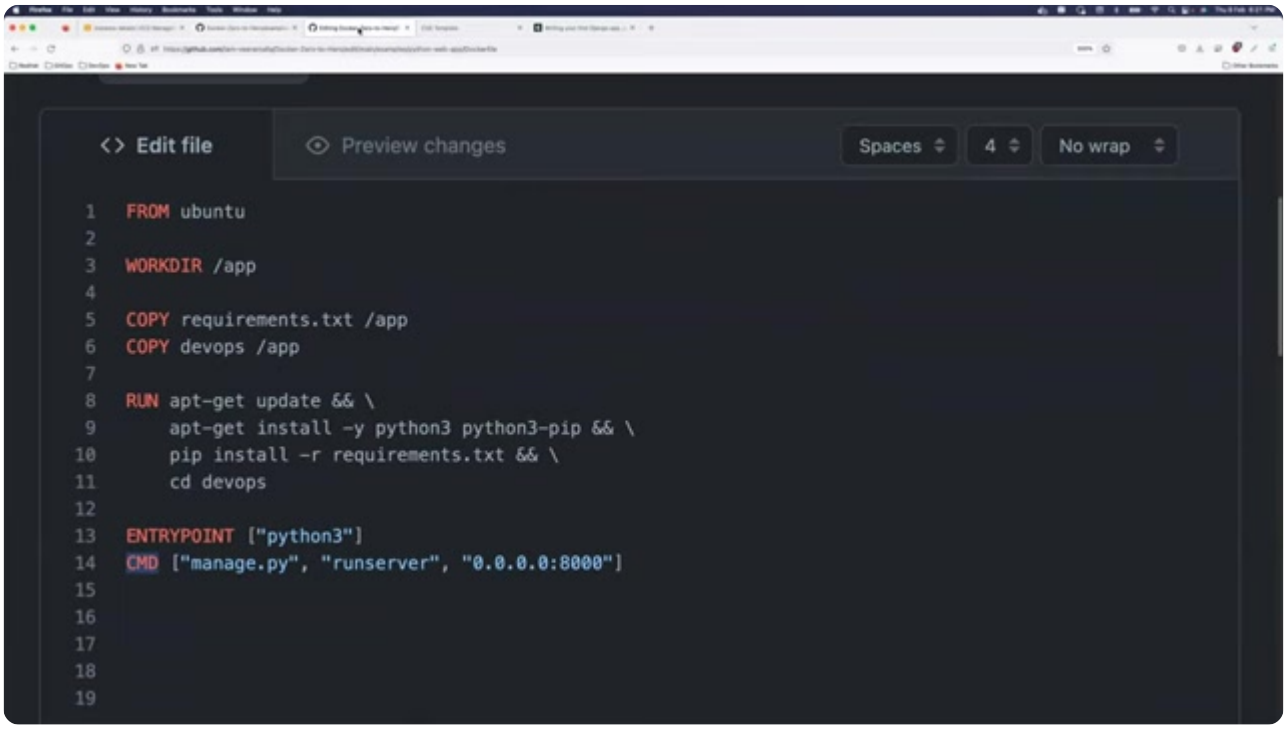
Now, let's build and run the container on our EC2 instance.

1. First, clone the repository: `git clone https://github.com/iam-veeramalla/Docker-Zero-to-Hero.git`
2. Navigate to the application directory: `cd Docker-Zero-to-Hero/examples/python-web-app`
3. Build the Docker image: `docker build .`
4. Verify the image was created: `docker images`
5. Run the container without port mapping: `docker run -it <image_id>`
    - At this point, the application is running inside the container, but it's not accessible from the host's browser because the container's port is not mapped to the host machine.

6. Run the container with port mapping: `docker run -p 8000:8000 -it <image_id>`

    - The `-p 8000:8000` flag maps port 8000 on the host to port 8000 in the container. Now, when you refresh your browser, the Django application loads successfully.

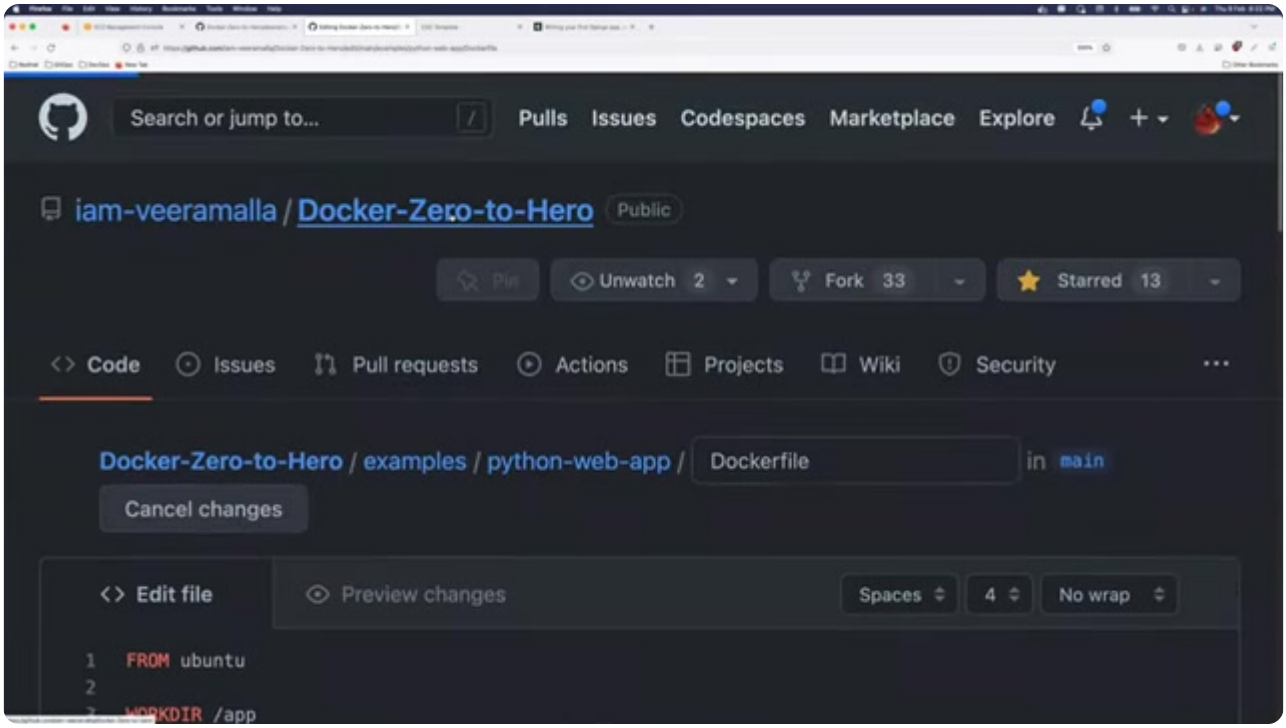
Exposing Ports in AWS Security Groups [00:25:36]



If you've done everything correctly and the application is still not accessible on your EC2 instance, the issue is likely with your AWS Security Group. You need to allow inbound traffic on the port your application is using.

1. Go to your EC2 instance in the AWS Console.
2. Click on the "Security" tab and then on your security group.
3. Click "Edit inbound rules" and "Add rule".
4. Set the type to "Custom TCP", the port range to "8000", and the source to "Anywhere" (0.0.0.0/0).
5. Save the rules. This will allow traffic to reach your application.

Conclusion and Next Steps [00:26:42]

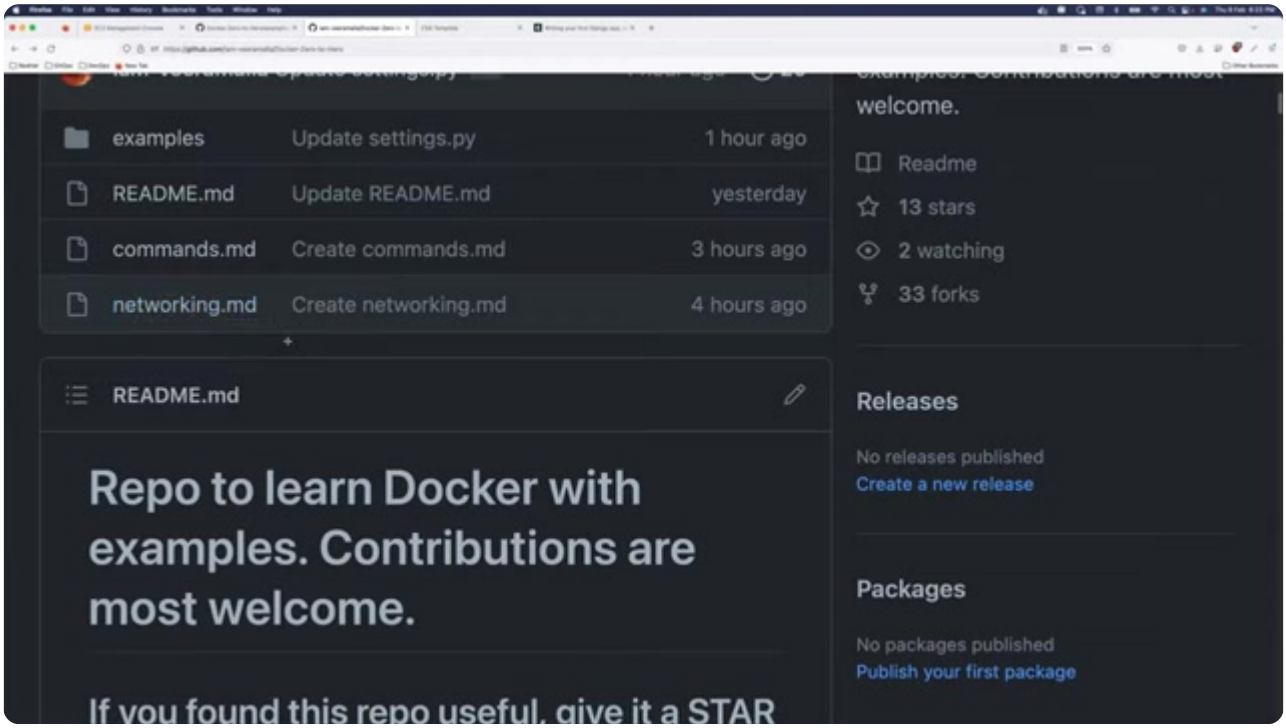


Today, we successfully containerized a Django web application. The most important step is to practice this yourself. Clone the repository, try to build and run the container, and even modify the `Dockerfile` to see what happens. This hands-on experience is key to building confidence.

In tomorrow's class, we will cover:

- More Docker commands.
- Docker networking.
- How to reduce container image size using multi-stage Docker builds.

A Bonus Question for Viewers [00:27:35]



Before I close, here is a question for you to answer in the comments: How many EC2 instances can you run on a free AWS account?

Thank you so much. I'll see you in the next video. Take care, everyone.