DevOps - Ideas

The Story of Susie who understands the power of DevOps

Meet Susie

Susie is a software engineer who just finished developing a web application.

- Now she needs to **deploy** it to a production server.

> Her manager says: "Just set up an NGINX server and deploy it!"

How hard could it be?

Chapter 1: The Manual Setup Journey

Susie starts setting up NGINX on her Ubuntu server...

```
# Step 1: Update the system
sudo apt update
sudo apt upgrade -y

# Step 2: Install NGINX
sudo apt install nginx -y

# Step 3: Install Node.js
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt install nodejs -y
```

Configuration Nightmare

```
# Step 4: Configure NGINX
sudo nano /etc/nginx/sites-available/myapp

# Step 5: Copy application files
scp -r ./myapp user@server:/var/www/myapp

# Step 6: Install dependencies
cd /var/www/myapp
npm install

# Step 7: Configure environment variables
sudo nano /etc/environment

# Step 8: Start the application
npm start
```

It took her 4 hours to start!

The Hidden Struggle

Even with Google, there are **countless variations** depending on:

- OS, distribution, version differences
- Undocumented configuration tricks and edge cases

Susie spent hours reading documentation alone.

Why alone? Most engineers don't consider configuration *"real engineering work"* — so she was hesitant to ask for help.

Chapter 2: Success... or is it?

After struggling with:

- Permission errors
- Configuration file syntax
- Port conflicts
- Environment variables
- Firewall settings

Finally, it works!

The application is running on the server!

The Deployment Disaster

One week later, Susie needs to deploy to **another server**...

```
# On the new server
sudo apt update
sudo apt install nginx -y
# Wait... which version of Node.js did I use?
# What was the NGINX configuration?
# Which environment variables did I set?
# What system packages did I install?
```

Oh no! The application doesn't work on the new server!

Why Did It Fail?

The new server has:

- Different OS version (Ubuntu 20.04 vs 22.04)
- Different Node.js version (v16 vs v18)
- Different system libraries
- Different default configurations

"It works on my machine!" syndrome

This is the classic deployment problem!

Chapter 3: Susie Discovers Docker

> Her colleague says: **"Why don't you use Docker?"**

Susie learns that Docker can:

- Package the application with all its dependencies
- Run the same way on **any machine**
- Make deployment reproducible and reliable

Let's see how!

The Application

First, Susie creates a simple Node.js application `hello.js`:

```
// hello.js
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Susie\'s Docker container!\n');
});

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Creating a Dockerfile

Susie creates a `Dockerfile` to containerize her app:

```
# Start from a base image with Node.js
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy application code
COPY hello.js .

# Expose port
EXPOSE 3000

# Start the application
CMD ["node", "hello.js"]
```

Understanding the Dockerfile

The `Dockerfile` is an **instruction manual** for Docker to:

1. **FROM node:18-alpine** - Start with Alpine Linux + Node.js 18 pre-installed
2. **WORKDIR /app** - Create and set working directory
3. **COPY hello.js .** - Copy the application file into the container
4. **EXPOSE 3000** - Document which port the app uses
5. **CMD ["node", "hello.js"]** - Run this command when container starts

This creates a self-contained "computer" with everything needed to run the app!

Running with Docker

Now Susie can deploy with simple commands:

```
# Build the Docker image
docker build -t susie-app .

# Run the container
docker run -p 3000:3000 susie-app
```

Visit http://localhost:3000 - It works!

This runs the SAME way on ANY machine with Docker!

Adding NGINX with Docker Compose

Now Susie wants to add NGINX as a reverse proxy. She creates `docker-compose.yml`:

```yaml
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - app
```

Understanding Docker Compose

docker-compose.yml orchestrates **multiple containers**:

- **app service**: Builds from Dockerfile (our custom app)
  - `build: .` means "use Dockerfile in current directory"
- **nginx service**: Uses pre-built image from Docker Hub
  - `image: nginx:alpine` means "download this ready-made image"

Why both?

- **Dockerfile** = Recipe for ONE custom container
- **docker-compose.yml** = Orchestrator for MULTIPLE containers working together

Deploying with Docker

Now deployment is **simple**:

```
# On ANY server (dev, staging, production)
docker-compose up -d
```

That's it!

- Works the same everywhere
- No manual configuration
- No dependency issues
- Repeatable and reliable

The Magic of Docker

Before Docker:

- 4 hours to set up one server
- Different results on different machines
- Hard to debug issues
- Difficult to scale

After Docker:

- 30 seconds to deploy
- Identical environment everywhere
- Easy to reproduce issues
- Simple to scale

Chapter 5: The DevOps Ecosystem

Susie learns there are **more DevOps tools**:

Containerization:

- Docker
- Kubernetes
- Podman

Infrastructure as Code:

- Terraform
- Ansible
- CloudFormation

CI/CD:

- GitHub Actions
- Jenkins
- GitLab CI

Monitoring:

- Prometheus
- Grafana
- ELK Stack

Chapter 6: Understanding DevOps Power

Susie now understands DevOps helps software engineers:

1. **Automate** repetitive tasks
2. **Standardize** environments
3. **Speed up** deployment
4. **Improve** reliability
5. **Scale** easily
6. **Collaborate** better

DevOps is not just tools, it's a mindset!

The DevOps Philosophy

Traditional Approach:

- Developers write code
- Operations team deploys
- Lots of manual work
- Slow and error-prone

DevOps Approach:

- Automate everything
- Infrastructure as code
- Continuous delivery
- Fast and reliable

Real-World Impact

Without DevOps:

- Deployment takes days
- High failure rate
- Difficult to rollback
- Manual testing

With DevOps:

- Deploy multiple times per day
- Automated testing
- Easy rollback
- Consistent quality