Docker & Jenkins DevOps - Part 2

A Complete Guide to Persistence, Networking, Registry & CI/CD

Table of Contents

Chapter 1: Docker Persistence & Volumes

Why Persistence Matters in Containers

Imagine you're running a MongoDB database in a Docker container. You spend hours setting up your database, adding users, creating collections, and populating it with data. Everything looks perfect! But then you restart the container, and suddenly... everything is gone. 😱

This happens because **containers are designed to be disposable**. When you stop and remove a container, everything inside its filesystem disappears forever. This is great for stateless applications like a React frontend, but terrible for databases or any application that needs to store data.

Understanding Container State

Stateless Applications:

- Frontend React builds
- Node.js API servers (without local file storage)
- Can be killed and redeployed anytime without losing important data

Stateful Applications:

- Databases (MongoDB, PostgreSQL, MySQL)
- File storage systems
- Applications that write logs or cache data
- Need their data to survive container restarts

What Are Docker Volumes?

A **Docker volume** is like an external hard drive for your containers. Think of it this way:

- **Without volumes**: Your data lives inside the container (like storing files only on your laptop's RAM - gone when you restart)
- **With volumes**: Your data lives outside the container in a persistent location (like storing files on an external hard drive - survives computer restarts)

Here's how it works:

1. Docker creates a storage area outside the container
2. The container "mounts" this storage area to a specific directory
3. When the container writes data to that directory, it actually goes to the volume
4. Even if you delete the container, the volume (and your data) remains

Volume Types in Docker

Named Volumes (Recommended)

```
# Create a named volume
docker volume create my-app-data

# Use it in a container
docker run -v my-app-data:/data/app my-app
```

Anonymous Volumes

```
# Docker creates a random name
docker run -v /data/app my-app
```

Bind Mounts

```
# Mount a specific host directory
docker run -v /host/path:/container/path my-app
```

Hands-On Volume Demo

Let's see persistence in action with a MongoDB container:

Step 1: Create a named volume

```
docker volume create demo-vol
docker volume ls
docker volume inspect demo-vol
```

Step 2: Run MongoDB with the volume

```
docker run -d --name demo-mongo -v demo-vol:/data/db mongo:6.0
```

Step 3: Add some data

```
docker exec -it demo-mongo mongosh --eval "
  use testdb;
  db.demos.insertOne({
    name:'docker-persist-demo',
    ts: new Date()
  })
"
```

Step 4: Verify the data exists

```
docker exec -it demo-mongo mongosh --eval "
  use testdb;
  db.demos.find().pretty()
"
```

Step 5: Delete the container (scary part!)

```
docker rm -f demo-mongo
```

Step 6: Create a new container with the same volume

```
docker run -d --name demo-mongo2 -v demo-vol:/data/db mongo:6.0
```

Step 7: Check if data survived

```
docker exec -it demo-mongo2 mongosh --eval "
  use testdb;
  db.demos.find().pretty()
"
```

 Magic! Your data is still there even though we completely destroyed the original container!

Why Volumes Are Essential for Production

Data Safety: Your database won't lose everything during updates
Backup Strategy: You can backup volumes independently of containers
Performance: Volumes often perform better than bind mounts
Portability: Move your app between environments without losing data
Security: Control exactly what data containers can access

Chapter 2: Docker Networking Deep Dive

The Container Isolation Challenge

By default, each Docker container is like a house with its own private address that nobody else knows about. This is great for security, but creates a problem: how do containers talk to each other?

Imagine you have a MERN application:

- Frontend container needs to call the backend API
- Backend container needs to connect to MongoDB
- Without proper networking, they can't find each other!

Docker Network Types

Bridge Network (Default)

- Every container automatically joins the default bridge
- Containers can talk to each other, but need IP addresses
- Not ideal for production applications

User-Defined Bridge (Recommended)

- You create a custom network
- Containers can find each other by name (like a phone book!)
- Much cleaner and more secure

Host Network

- Container shares the host's network directly
- No isolation, but sometimes needed for performance
- Linux only

None Network

- No networking at all
- Used for security-sensitive applications

Container Name Resolution (The Magic of DNS)

Here's the coolest part about user-defined networks: **container names become like website addresses!**

In a custom network called `mern-net`:

- Backend container named `backend`
- MongoDB container named `mongo`
- Frontend can call: `http://backend:5000/api`
- Backend can connect to: `mongodb://mongo:27017/mydb`

Docker automatically handles the "phone book lookup" - no need to remember IP addresses!

Networking Commands

```
# List all networks
docker network ls

# Create a custom network
docker network create mern-net

# Inspect network details
docker network inspect mern-net

# Remove a network
docker network rm mern-net
```

Real-World Networking Example

Let's create a mini web application with proper networking:

Step 1: Create a custom network

```
docker network create app-net
```

Step 2: Run a simple web server

```
docker run -d --name web-server --network app-net nginx
```

Step 3: Run a client that can talk to the server

```
docker run --rm --network app-net curlimages/curl curl http://web-server
```

Notice how we used `web-server` as the hostname, not an IP address!

Why Custom Networks Matter

Security: Only containers in the same network can talk to each other
Simplicity: Use container names instead of remembering IP addresses
Reliability: IP addresses change when containers restart, names don't
Organization: Separate different applications into different networks

Chapter 3: Docker Registry Fundamentals

The Image Sharing Problem

You've built an amazing Docker image on your laptop. Now you want to:

- Share it with your teammates
- Deploy it on a server
- Use it in your CI/CD pipeline

But Docker images only exist locally on the machine where you built them. How do you share them with the world?

What is a Docker Registry?

A **Docker registry** is like GitHub for Docker images. Just as GitHub stores and shares code repositories, a registry stores and shares Docker images.

Popular Registries:

- **Docker Hub**: Free public registry (like GitHub public repos)
- **GitHub Container Registry**: Integrated with GitHub
- **Amazon ECR**: Amazon's private registry
- **Google Container Registry**: Google's registry service

The Image Sharing Workflow

Here's how image sharing works in the real world:

```
Developer's Laptop    →    Docker Registry    →    Production Server
     |                          |                          |
1. docker build         2. docker push           3. docker pull
2. docker tag           3. Images stored          4. docker run
3. docker push             safely in cloud
```

Hands-On Registry Demo

Let's push an image to Docker Hub and pull it back:

Step 1: Build a simple image

```
# Create a simple Dockerfile
echo 'FROM nginx
COPY . /usr/share/nginx/html' &gt; Dockerfile

echo '<h1>Hello from Docker Registry!</h1>' &gt; index.html

docker build -t my-web-app .
```

Step 2: Tag for Docker Hub

```
# Format: username/repository:tag
docker tag my-web-app yourusername/my-web-app:1.0.0
```

Step 3: Login and push

```
docker login
docker push yourusername/my-web-app:1.0.0
```

Step 4: Simulate another machine

```
# Remove local copy
docker rmi my-web-app yourusername/my-web-app:1.0.0

# Pull from registry
docker pull yourusername/my-web-app:1.0.0

# Run the pulled image
docker run -p 8080:80 yourusername/my-web-app:1.0.0
```

Registry Best Practices

Image Tagging Strategy:

- Use semantic versioning: `1.0.0`, `1.1.0`, `2.0.0`
- Tag stable releases: `latest`, `stable`
- Include build numbers: `v1.0.0-build.123`

Security Considerations:

- Use private registries for proprietary code
- Scan images for vulnerabilities
- Use minimal base images (Alpine Linux)

Chapter 4: MERN Implementation with Docker

Now let's put everything together! We'll take a complete MERN application and dockerize it with proper persistence, networking, and registry integration.

Application Architecture

Our MERN stack will have:

- **MongoDB**: Database with persistent volume
- **Express/Node Backend**: API server
- **React Frontend**: User interface
- **Custom Network**: So containers can communicate
- **Registry Integration**: Push/pull images for deployment

Setting Up the Environment

Step 1: Update Backend Configuration

Edit `server/.env`:

```
PORT=5000
MONGO_URI=mongodb://mongo:27017/taskdb
```

Notice we're using `mongo` as the hostname - this will be our MongoDB container name!

Step 2: Create Docker Compose Configuration

Create `docker-compose.yml` in your project root:

```
services:
  mongo:
    image: mongo:6.0
    container_name: mongo
    restart: unless-stopped
    volumes:
      - mongo-data:/data/db
    networks:
      - mern-net

  backend:
    build:
      context: ./server
      dockerfile: Dockerfile
    container_name: backend
    restart: unless-stopped
    env_file:
      - ./server/.env
    ports:
      - "5000:5000"
    depends_on:
      - mongo
    networks:
      - mern-net

  frontend:
    build:
      context: ./client
      dockerfile: Dockerfile
      args:
        VITE_API_URL: http://localhost:5000/api
    container_name: frontend
    restart: unless-stopped
    ports:
      - "5173:5173"
    depends_on:
      - backend
    networks:
      - mern-net

volumes:
  mongo-data:

networks:
  mern-net:
    driver: bridge
```

Running the Complete Application

Step 1: Build and start everything

```
docker compose up --build -d
```

Step 2: Verify all containers are running

```
docker ps
```

You should see three containers: mongo, backend, and frontend.

Step 3: Test the application

- Open http://localhost:5173 for the frontend
- Test API directly: http://localhost:5000/api/tasks

Testing Data Persistence

Let's verify our MongoDB data survives container restarts:

Step 1: Create some data

```
curl -X POST http://localhost:5000/api/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Learn Docker Volumes", "completed": false}'
```

Step 2: Verify data exists

```
curl http://localhost:5000/api/tasks
```

Step 3: Restart the backend (simulating deployment)

```
docker compose stop backend
docker compose rm -f backend
docker compose up -d backend
```

Step 4: Check data survived

```
curl http://localhost:5000/api/tasks
```

 Your data should still be there because MongoDB's data is stored in the persistent volume!

Pushing Images to Registry

Now let's prepare our images for production deployment:

Step 1: Build production images

```
docker build -t mern-backend:local ./server
docker build -t mern-frontend:local ./client \
  --build-arg VITE_API_URL=http://localhost:5000/api
```

Step 2: Tag for Docker Hub

```
docker tag mern-backend:local yourusername/mern-backend:1.0.0
docker tag mern-frontend:local yourusername/mern-frontend:1.0.0
```

Step 3: Push to registry

```
docker login
docker push yourusername/mern-backend:1.0.0
docker push yourusername/mern-frontend:1.0.0
```

Step 4: Test deployment from registry

```
# Stop local containers
docker compose down

# Remove local images
docker rmi mern-backend:local mern-frontend:local

# Pull from registry and run
docker pull yourusername/mern-backend:1.0.0
docker pull yourusername/mern-frontend:1.0.0

# Run with registry images
docker network create mern-net
docker run -d --name mongo --network mern-net -v mongo-data:/data/db mongo:6.0
docker run -d --name backend --network mern-net \
  -e MONGO_URI='mongodb://mongo:27017/taskdb' \
  -p 5000:5000 yourusername/mern-backend:1.0.0
docker run -d --name frontend --network mern-net \
  -p 5173:5173 yourusername/mern-frontend:1.0.0
```

Understanding the Network Flow

Here's how the containers communicate:

1. **Browser → Frontend**: User visits http://localhost:5173
2. **Frontend → Backend**: JavaScript calls http://localhost:5000/api
3. **Backend → MongoDB**: Server connects to mongodb://mongo:27017

The magic is that `mongo` resolves to the MongoDB container's IP address within the `mern-net` network!

Chapter 5: Jenkins Introduction & Setup

What Problems Does Jenkins Solve?

Imagine your daily development workflow without automation:

1. Write code and commit to GitHub ✅
2. Manually clone the repo on the server 😰
3. Manually run `npm install` for backend and frontend 😰
4. Manually build Docker images 😰
5. Manually push images to registry 😰
6. Manually run `docker compose up` 😰
7. Manually test that everything works 😰

This process is:

- **Error-prone**: Easy to forget steps or make mistakes
- **Time-consuming**: Takes 20+ minutes each time
- **Inconsistent**: Different team members might do things differently
- **Not scalable**: What if you need to deploy 10 times per day?

Enter Jenkins: Your DevOps Robot

Jenkins is like having a tireless robot assistant that can:

- Watch your GitHub repository 24/7
- Automatically run your build process when you push code
- Test your application and notify you of problems
- Deploy successful builds to production
- Keep detailed logs of everything it does

Think of Jenkins as a factory manager that follows your exact instructions, every single time, without getting tired or making human errors.

Key Jenkins Concepts

Job (Project): A set of instructions for Jenkins to follow

- Example: "Build my MERN app and run tests"

Build: One execution of a job

- Each time Jenkins runs your job, it gets a number (#1, #2, #3...)

Pipeline: A job defined as code (Jenkinsfile)

- Like a recipe that Jenkins follows step by step

Agent/Node: The computer where Jenkins runs your jobs

- Can be the Jenkins server itself or other connected machines

Workspace: A temporary folder where Jenkins works

- Contains your source code and build artifacts

Trigger: What causes a job to run

- Manual click, GitHub push, scheduled time, etc.

Installing Jenkins with Docker

We'll run Jenkins in Docker for consistency and easy setup:

Step 1: Pull the Jenkins image

```
docker pull jenkins/jenkins:lts
```

Step 2: Run Jenkins container

```
docker run -d \
  --name jenkins \
  -p 8080:8080 -p 50000:50000 \
  -v jenkins_home:/var/jenkins_home \
  jenkins/jenkins:lts
```

What this does:

- `-p 8080:8080`: Jenkins web interface on http://localhost:8080
- `-p 50000:50000`: Port for Jenkins agents (advanced usage)
- `-v jenkins_home:/var/jenkins_home`: Persistent storage for Jenkins data

Step 3: Get the initial admin password

```
docker logs jenkins
```

Look for a message like:

```
Jenkins initial setup is required.
Admin password: 1234567890abcdef...
```

Or get it directly:

```
docker exec jenkins cat /var/jenkins_home/secrets/initialAdminPassword
```

Step 4: Complete Jenkins setup

1. Open http://localhost:8080
2. Enter the admin password
3. Choose "Install suggested plugins"

4. Create an admin user account
5. Set Jenkins URL (default is fine)

□ Jenkins is now ready!

Jenkins Dashboard Walkthrough

When you log into Jenkins, you'll see several key areas:

Main Dashboard

- Shows all your jobs and their status
- Green = success, Red = failure, Yellow = unstable

New Item

- Create new jobs (Freestyle or Pipeline)

Manage Jenkins

- Install plugins, configure security, manage nodes
- Think of this as Jenkins' "Settings" menu

Build History

- Shows recent job executions across all projects

Credentials

- Securely store passwords, API keys, SSH keys
- Never hardcode secrets in your job configurations!

Jenkins for MERN Development

Here's how Jenkins will help with your MERN application:

Before Jenkins (Manual Process):

1. Developer pushes code □ 2 minutes
2. Manually pull code □ 1 minute
3. Install dependencies □ 3 minutes
4. Build Docker images □ 5 minutes
5. Push to registry □ 2 minutes
6. Deploy and test □ 5 minutes
   Total: 18 minutes of manual work, every single time

With Jenkins (Automated Process):

1. Developer pushes code □ 2 minutes
2. Jenkins automatically does steps 2-6 □ 10 minutes
3. Jenkins sends notification when done □ 0 minutes
   Total: 2 minutes of developer time, more consistent results

Chapter 6: Jenkins Jobs & Pipelines

Starting Simple: Your First Jenkins Job

Let's create a basic job to understand how Jenkins works before diving into complex MERN automation.

Creating a "Hello World" Job:

1. **Go to Jenkins Dashboard** → Click "New Item"
2. **Enter Job Name**: `hello-jenkins`
3. **Select**: Freestyle project → Click OK
4. **Add Build Step**: Execute shell
5. **Enter Command**:

   ```
   echo "Hello Jenkins! □"
   echo "Current date: $(date)"
   echo "Jenkins workspace: $WORKSPACE"
   ```

6. **Save** → Click "Build Now"

Check the Results:

- Look at "Build History" → Click build #1
- Click "Console Output" to see what happened

You should see output like:

```
Started by user admin
Building in workspace /var/jenkins_home/workspace/hello-jenkins
+ echo Hello Jenkins! □
Hello Jenkins! □
+ echo Current date: Tue Sep 16 19:30:00 UTC 2025
Current date: Tue Sep 16 19:30:00 UTC 2025
+ echo Jenkins workspace: /var/jenkins_home/workspace/hello-jenkins
Jenkins workspace: /var/jenkins_home/workspace/hello-jenkins
Finished: SUCCESS
```

Understanding Job Parameters

Sometimes you want jobs to behave differently based on input. Let's create a parameterized job:

Creating a Parameterized Job:

1. **New Item** → `deploy-environment` → Freestyle project
2. **Check**: "This project is parameterized"
3. **Add Parameter**: String Parameter
   - Name: `ENVIRONMENT`
   - Default: `development`
   - Description: `Target environment (development, staging, production)`

4. **Build Step**: Execute shell

   ```
   echo "Deploying to environment: $ENVIRONMENT"
   echo "Deployment started at: $(date)"

   if [ "$ENVIRONMENT" = "production" ]; then
       echo "□ PRODUCTION DEPLOYMENT - Extra checks required!"
   else
       echo "✅ $ENVIRONMENT deployment - proceeding normally"
   fi
   ```

5. **Save** → Click "Build with Parameters"

Now Jenkins will ask you which environment to deploy to before running!

Connecting Jenkins to GitHub

Real projects live in version control. Let's connect Jenkins to a GitHub repository:

Prerequisites:

- A GitHub repository with your MERN project
- Repository can be public (easier) or private (requires credentials)

Creating a GitHub-Connected Job:

1. **New Item** → `mern-github-demo` → Freestyle project
2. **Source Code Management** → Select "Git"
3. **Repository URL**: `https://github.com/yourusername/your-mern-repo.git`
4. **Branch**: `*/main` (or `*/master`)
5. **Build Steps** → Execute shell:

   ```
   echo "Repository cloned successfully!"
   echo "Repository contents:"
   ls -la

   echo "Checking for package.json files:"
   find . -name "package.json" -type f

   echo "Server dependencies:"
   if [ -f "server/package.json" ]; then
       cat server/package.json | grep -A 10 '"dependencies"'
   fi
   ```

Understanding What Happens:

1. Jenkins clones your GitHub repo into its workspace
2. Runs your shell commands in that directory
3. Shows you the repository contents and structure

Pipeline vs Freestyle Jobs

Freestyle Jobs:

- ✅ Easy to create with GUI
- ✅ Good for simple tasks
- ❌ Hard to version control
- ❌ Limited flexibility for complex workflows

Pipeline Jobs:

- ✅ Defined as code (Jenkinsfile)
- ✅ Version controlled with your application
- ✅ Support complex workflows with multiple stages
- ✅ Can be reviewed and collaborated on like regular code
- ❌ Requires learning Pipeline syntax

Introduction to Pipeline Jobs

A Pipeline job reads a special file called `Jenkinsfile` from your repository. This file contains the instructions for building your application.

Basic Pipeline Structure:

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                echo 'Getting source code...'
            }
        }

        stage('Build') {
            steps {
                echo 'Building application...'
            }
        }

        stage('Test') {
            steps {
                echo 'Running tests...'
            }
        }

        stage('Deploy') {
            steps {
                echo 'Deploying application...'
            }
        }
    }
}
```

Creating Your First Pipeline Job:

1. **New Item** → `mern-pipeline-demo` → Pipeline
2. **Pipeline Definition** → "Pipeline script"
3. **Copy the basic pipeline structure above**
4. **Save** → **Build Now**

You'll see each stage appear as a separate box in the Jenkins interface, making it easy to see which step succeeded or failed.

Job Triggers: When Should Jenkins Run?

Manual Trigger: Click "Build Now"

- Good for: Testing, one-off deployments

SCM Polling: Jenkins checks GitHub periodically

```
H/5 * * * *  # Check every 5 minutes
```

- Good for: When webhooks aren't available

GitHub Webhooks: GitHub notifies Jenkins immediately

- Good for: Production systems (fastest response)

Scheduled Builds: Run at specific times

```
H 2 * * *    # Every night at 2 AM
H H * * 1    # Every Monday
```

- Good for: Nightly builds, regular maintenance

Building Toward MERN Automation

Now that you understand Jenkins basics, here's how we'll build up to a complete MERN CI/CD pipeline:

Stage 1: Simple GitHub integration ✅ (Just completed)
Stage 2: Build Docker images in Jenkins
Stage 3: Run and test the complete MERN stack
Stage 4: Push images to Docker registry
Stage 5: Deploy to production environment

Each stage builds on the previous one, so you always have a working foundation to fall back on.

Chapter 7: Complete MERN CI/CD Pipeline

Pipeline Overview

Our complete CI/CD pipeline will automate the entire journey from code commit to running application:

1. **Trigger**: Developer pushes code to GitHub
2. **Checkout**: Jenkins pulls the latest code
3. **Build**: Create Docker images for backend and frontend
4. **Test**: Verify the application works correctly
5. **Deploy**: Run the complete MERN stack
6. **Cleanup**: Clean up resources for the next build

Prerequisites Setup

Before creating the pipeline, we need Jenkins to be able to control Docker:

Option 1: Docker Socket Mount (Development)

```
docker run -d \
  --name jenkins \
  -p 8080:8080 -p 50000:50000 \
  -v jenkins_home:/var/jenkins_home \
  -v /var/run/docker.sock:/var/run/docker.sock \
  jenkins/jenkins:lts
```

Option 2: Docker-in-Docker (Production)
Use the official Jenkins Docker image with Docker pre-installed.

The Complete Jenkinsfile

Create this file as `Jenkinsfile` in your repository root:

```
pipeline {
    agent any

    environment {
        // Define image names as variables for easy maintenance
        BACKEND_IMAGE = "mern-backend:jenkins"
        FRONTEND_IMAGE = "mern-frontend:jenkins"
        DOCKER_REGISTRY = "your-dockerhub-username"
    }

    stages {
        stage('Checkout Code') {
            steps {
                echo ' Checking out source code from GitHub...'
                // Jenkins automatically clones the repo containing this Jenkinsfile
                script {
                    echo "Building commit: ${env.GIT_COMMIT}"
                    echo "Branch: ${env.GIT_BRANCH}"
                }
            }
        }

        stage('Build Docker Images') {
            steps {
                echo '□ Building Docker images...'
                script {
                    // Build backend image
                    echo "Building backend image: ${BACKEND_IMAGE}"
                    sh "docker build -t ${BACKEND_IMAGE} ./server"

                    // Build frontend image with API URL
                    echo "Building frontend image: ${FRONTEND_IMAGE}"
                    sh """
                        docker build -t ${FRONTEND_IMAGE} ./client \
                        --build-arg VITE_API_URL=http://localhost:5000/api
                    """

                    // Verify images were created
                    sh "docker images | grep jenkins"
                }
            }
        }

        stage('Start Application Services') {
            steps {
                echo '□ Starting MERN application...'
                script {
                    // Start the complete application stack
                    sh 'docker compose up -d'

                    // Wait for services to be ready
                    echo "Waiting for services to start..."
                    sleep(time: 30, unit: "SECONDS")

                    // Show running containers
                    sh 'docker ps --format "table {{.Names}}\\t{{.Image}}\\t{{.Status}}\\t
                }
            }
        }

        stage('Health Check &amp; Testing') {
            steps {
                echo '□ Running health checks...'
                script {
                    // Test backend API
                    echo "Testing backend API..."
                    sh '''
                        curl -f http://localhost:5000/health || {
                            echo "❌ Backend health check failed"
                            exit 1
                        }
                        echo "✅ Backend is healthy"
                    '''
                    // Test frontend
                    echo "Testing frontend..."
                    sh '''
                        curl -f http://localhost:5173 || {
                            echo "❌ Frontend health check failed"
                            exit 1
                        }
                        echo "✅ Frontend is accessible"
                    '''

                    // Test database connection through API
                    echo "Testing database connectivity..."
                    sh '''
                        curl -f http://localhost:5000/api/tasks || {
                            echo "❌ Database connection failed"
                            exit 1
                        }
                        echo "✅ Database is connected"
                    '''
                }
            }
        }

        stage('Integration Tests') {
            steps {
                echo '□ Running integration tests...'
                script {
                    // Create a test task
                    sh '''
                        echo "Creating test task..."
                        curl -X POST http://localhost:5000/api/tasks \
                             -H "Content-Type: application/json" \
                             -d '{"title": "Jenkins CI Test Task", "completed": false}' \
                             -f || exit 1
                    '''

                    // Verify task was created
                    sh '''
                        echo "Verifying task creation..."
                        curl http://localhost:5000/api/tasks | grep "Jenkins CI Test Task"
                            echo "❌ Task creation test failed"
                            exit 1
                        }
                        echo "✅ Integration test passed"
                    '''
                }
            }
        }

        stage('Performance Check') {
            steps {
                echo '⚡ Running basic performance checks...'
                script {
                    // Simple response time check
                    sh '''
                        echo "Checking API response time..."
                        time curl -s http://localhost:5000/api/tasks &gt; /dev/null

                        echo "Checking frontend load time..."
                        time curl -s http://localhost:5173 &gt; /dev/null
                    '''
                }
            }
        }
    }

    post {
        always {
            echo '▯ Cleaning up resources...'
            script {
                // Always clean up, regardless of build result
                sh '''
                    echo "Stopping application containers..."
                    docker compose down || true

                    echo "Removing test containers..."
                    docker rm -f $(docker ps -aq --filter "label=jenkins-test") || true

                    echo "Cleaning up unused images..."
                    docker image prune -f || true
                '''
            }
        }

        success {
            echo '✅ Pipeline completed successfully!'
            script {
                // Additional success actions
                sh 'echo "Build #${BUILD_NUMBER} succeeded at $(date)"'
            }
        }

        failure {
            echo '❌ Pipeline failed!'
            script {
                // Capture logs for debugging
                sh '''
                    echo "Capturing container logs for debugging..."
                    docker compose logs || true
                '''
            }
        }

        unstable {
            echo '⚠ Pipeline completed with warnings'
        }
    }
}
```

Understanding the Pipeline Flow

Stage Breakdown:

1. **Checkout Code**
    - Jenkins automatically pulls your GitHub repository
    - Shows commit hash and branch information
    - Sets up the workspace for building
2. **Build Docker Images**
    - Creates backend image from `./server/Dockerfile`
    - Creates frontend image with production API URL
    - Verifies images were built successfully
3. **Start Application Services**
    - Runs `docker compose up -d` to start all services
    - Waits for containers to fully initialize
    - Shows running container status
4. **Health Check & Testing**
    - Tests each service individually
    - Verifies API endpoints respond correctly
    - Ensures database connectivity works
5. **Integration Tests**
    - Creates actual test data through the API
    - Verifies end-to-end functionality
    - Tests the complete request/response cycle
6. **Performance Check**
    - Basic response time verification
    - Ensures services are performing adequately
    - Can be extended with more sophisticated metrics

Setting Up the Pipeline Job

Step 1: Create the Pipeline Job

1. Jenkins Dashboard → New Item
2. Name: `mern-ci-cd-pipeline`
3. Type: Pipeline → OK

Step 2: Configure the Pipeline

1. **General**: Add description "Complete MERN CI/CD Pipeline"

2. **Build Triggers**: Check "GitHub hook trigger for GITScm polling"
3. **Pipeline**:
    - Definition: "Pipeline script from SCM"
    - SCM: Git
    - Repository URL: Your GitHub repo URL
    - Branch: `*/main`
    - Script Path: `Jenkinsfile`

Step 3: Set Up GitHub Webhook (Optional but Recommended)

1. Go to your GitHub repository
2. Settings → Webhooks → Add webhook
3. Payload URL: `http://your-jenkins-url:8080/github-webhook/`
4. Content type: `application/json`
5. Events: "Just the push event"

Running and Monitoring the Pipeline

First Run:

1. Save the pipeline configuration
2. Click "Build Now" for the first manual run
3. Watch the stage progress in real-time

Understanding the Build View:

- **Blue bars**: Stages in progress
- **Green bars**: Successful stages
- **Red bars**: Failed stages
- **Console Output**: Detailed logs for debugging

Monitoring Tips:

- Click on individual stages to see their specific output
- Use "Console Output" for complete build logs
- Check "Build History" for trends over time

Extending the Pipeline

Adding Registry Push:

```
stage('Push to Registry') {
    when {
        branch 'main'  // Only push from main branch
    }
    steps {
        script {
            withCredentials([usernamePassword(credentialsId: 'dockerhub-creds',
                                             usernameVariable: 'USERNAME',
                                             passwordVariable: 'PASSWORD')]) {
                sh 'docker login -u $USERNAME -p $PASSWORD'
                sh "docker tag ${BACKEND_IMAGE} ${DOCKER_REGISTRY}/mern-backend:${BUILD_NU
                sh "docker tag ${FRONTEND_IMAGE} ${DOCKER_REGISTRY}/mern-frontend:${BUILD_
                sh "docker push ${DOCKER_REGISTRY}/mern-backend:${BUILD_NUMBER}"
                sh "docker push ${DOCKER_REGISTRY}/mern-frontend:${BUILD_NUMBER}"
            }
        }
    }
}
```

Adding Slack Notifications:

```
post {
    success {
        slackSend(
            channel: '#deployments',
            color: 'good',
            message: "✅ MERN Pipeline #${BUILD_NUMBER} succeeded! 🚀"
        )
    }
    failure {
        slackSend(
            channel: '#deployments',
            color: 'danger',
            message: "❌ MERN Pipeline #${BUILD_NUMBER} failed! Check logs: ${BUILD_URL}"
        )
    }
}
```

Troubleshooting Common Issues

"Docker command not found"

- Ensure Jenkins can access Docker (socket mount or Docker-in-Docker)
- Verify Docker is installed on the Jenkins agent

"Port already in use"

- Modify docker-compose.yml to use different ports for Jenkins builds
- Or ensure cleanup happens between builds

"Permission denied"

- Jenkins user needs Docker group membership
- Check file permissions in the workspace

"GitHub webhook not triggering"

- Verify webhook URL is correct and accessible
- Check Jenkins GitHub plugin configuration

- Look for webhook delivery logs in GitHub

Production Considerations

Security:

- Use Jenkins credentials for sensitive data
- Don't hardcode passwords in Jenkinsfiles
- Limit Jenkins network access

Performance:

- Use dedicated build agents for heavy workloads
- Implement build caching strategies
- Parallelize independent stages

Reliability:

- Add retry logic for flaky network operations
- Implement proper error handling
- Set reasonable timeouts for all operations

Monitoring:

- Set up build failure notifications
- Monitor build times and success rates
- Log important metrics for analysis

Conclusion

You've now learned how to create a complete DevOps pipeline that automates your MERN application from development to deployment. This pipeline includes:

✅ **Persistent Data Storage** with Docker volumes
✅ **Container Networking** for service communication
✅ **Image Registry** integration for deployment
✅ **Automated CI/CD** with Jenkins pipelines
✅ **Automated Testing** and health checks
✅ **Production-Ready** practices and configurations

Next Steps

1. **Implement the pipeline** in your own MERN project
2. **Add more sophisticated tests** (unit tests, E2E tests)
3. **Explore advanced Jenkins features** (parallel stages, matrix builds)
4. **Learn about production deployment** (Kubernetes, AWS, cloud platforms)
5. **Implement monitoring and logging** (Prometheus, Grafana, ELK stack)

Remember: DevOps is a journey, not a destination. Start with this foundation and continuously improve your pipeline as you learn more about your application's needs and your team's workflow.

Happy building!