Prepared By:
DEVOPS SHACK

DOCKER

Real-World Scenarios with Solutions

E-BOOK 2025

www.devopsshack.com

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Mastering Docker: Real-World Scenarios with Solutions

Table of Contents

18. Rolling back to a previous Docker image version.
19. Migrating data between Docker volumes.
20. Monitoring logs across multiple containers.

21–30: Network, Security, and Resource Management

21. Configuring specific DNS servers for containers.
22. Restricting access to a container on a specific IP.
23. Configuring container logging.
24. Restarting containers automatically with delay.
25. Preventing data loss during container recreation.
26. Managing multiple versions of Docker Compose files.
27. Scaling services in Docker Compose.
28. Exposing Docker services securely to the internet.
29. Inspecting the filesystem of a running container.
30. Cleaning up unused Docker objects.

31–40: Operational Challenges and Optimization

31. Running a container with a specific DNS configuration.
32. Restricting access to containers on specific IPs.
33. Handling containers consuming too much memory.
34. Running a container in detached mode and reattaching later.
35. Limiting the number of processes inside a container.
36. Restarting containers with updated environment variables.
37. Ensuring containers start automatically on system boot.
38. Securing containers running sensitive applications.
39. Connecting a container to a running database container.
40. Versioning Docker images in CI/CD pipelines.

41–50: Advanced Scenarios, Security, and Deployment

41. Running a container in privileged mode.
42. Managing multiple containers with different configurations in Docker Compose.
43. Debugging slow container startup times.
44. Cleaning up exited containers automatically.
45. Implementing health checks in containers.
46. Backing up and restoring Docker volumes.
47. Deploying a multi-tier application using Docker Compose.
48. Handling secrets securely in Docker Compose.
49. Running containers with GPU support.
50. Securing Docker images before deployment.

Introduction to Docker Scenarios

Docker has become a cornerstone technology for modern software development and deployment, enabling teams to build, ship, and run applications in consistent, portable environments. However, working with Docker in real-world scenarios often presents challenges that go beyond basic usage. From debugging failing containers to implementing secure deployments, these challenges require a solid understanding of Docker's features and best practices.

This document presents 50 scenario-based questions and answers to help developers, DevOps engineers, and system administrators navigate common and advanced Docker challenges. Each scenario is designed to replicate real-world issues and provides detailed solutions to equip you with the knowledge needed to troubleshoot, optimize, and scale containerized applications effectively.

The scenarios are categorized into themes such as debugging, resource management, security, network configuration, and deployment strategies. Whether you're a beginner looking to strengthen your foundational skills or an experienced practitioner seeking advanced tips, this collection will guide you through the intricacies of Docker.

By exploring these scenarios, you will:

- Learn to troubleshoot and optimize containerized environments.
- Gain practical insights into handling Docker-related complexities in production.
- Understand best practices for secure and efficient Docker usage.

Dive into these scenarios to enhance your expertise and ensure your Docker workflows are robust, scalable, and ready to tackle real-world challenges.

Basic Debugging and Optimization

1. How do you debug a container that is failing to start?

Answer:

- Use the docker logs command to check the container's logs:

```
docker logs <container_id>
```

- If the container exits immediately, use the --tail and --follow options to monitor the logs.
- Start the container in an interactive shell to investigate:

```
docker run -it <image_name> /bin/bash
```

- Check the Dockerfile for potential misconfigurations like incorrect entry points.
- Use docker inspect <container_id> to check the container's configuration.
- Debug startup commands using CMD or ENTRYPOINT by overriding them:

```
docker run -it <image_name> /bin/bash
```

2. What happens if you bind a volume that doesn't exist on the host?

Answer:

- If the volume path doesn't exist, Docker will automatically create a directory at the specified path on the host.
- Example:

```
docker run -v /nonexistent:/app alpine
```

This will create /nonexistent on the host system.

- The default permissions will depend on the user running Docker.
- It is a best practice to ensure the directory exists with proper permissions before running the container.

3. How do you reduce the size of a Docker image?

Answer:

1. Use smaller base images like alpine:

```
FROM alpine:latest
```

2. Minimize the number of layers by combining commands:

```
RUN apt-get update && apt-get install -y \
  curl \
  vim && \
  apt-get clean
```

3. Remove unnecessary files in the same layer:

```
RUN rm -rf /var/lib/apt/lists/*
```

4. Use .dockerignore to exclude files from the build context.

4. How do you pass sensitive data to a Docker container securely?

Answer:

- Use Docker secrets:
    1. Create a secret:

```
echo "my-secret-password" | docker secret create db_password -
```

    2. Use the secret in a Docker service:

```
docker service create --name my-service --secret db_password alpine
```

- Pass environment variables securely with docker-compose:

```
version: '3.7'
services:
 app:
  image: my-app
  environment:
   - SECRET_KEY=${SECRET_KEY}
```

Use .env to store sensitive values and ensure it's excluded in .gitignore.

5. How do you troubleshoot network issues between Docker containers?

Answer:

1. Check the network configuration:

```
docker network inspect <network_name>
```

2. Verify container connectivity using ping:

```
docker exec <container_id> ping <another_container>
```

3. Ensure both containers are on the same network.
4. Examine firewall rules and port mappings.
5. Use curl or telnet to test specific ports:

```
docker exec <container_id> curl http://<other_container>:<port>
```

6. How do you handle a container running out of disk space?

Answer:

- Check disk usage:

```
docker system df
```

- Prune unused resources:

```
docker system prune -a --volumes
```

- Clean unused images:

```
docker image prune -a
```

- Mount volumes to avoid writing large files in the container's filesystem.

7. How do you ensure a container restarts automatically if it crashes?

Answer:

- Use the --restart policy:

```
docker run --restart always <image_name>
```

- Available restart policies:
    - no: Default, no automatic restart.

- on-failure: Restart only if the container exits with a non-zero status.
- always: Restart regardless of exit status.
- unless-stopped: Restart unless the container is manually stopped.

8. How do you update an application running in a container without downtime?

Answer:

- Use a rolling update strategy with docker-compose:

```
version: '3.9'
services:
  app:
    image: my-app:latest
    deploy:
      update_config:
        parallelism: 2
        delay: 10s
```

- Replace the running container:

```
docker-compose up -d --no-deps --build app
```

9. How do you debug a build failure in a Dockerfile?

Answer:

1. Use incremental builds:

```
docker build --target <stage_name> .
```

2. Add debugging steps to inspect intermediate layers:

```
RUN ls -al /path/to/debug
```

3. Use docker history to examine the image layers:

```
docker history <image_id>
```

4. Run problematic commands interactively in a base image:

```
docker run -it <base_image> /bin/bash
```

10. How do you persist data in a stateless Docker container?

Answer:

- Use volumes to persist data:

```
docker run -v /host/data:/container/data <image_name>
```

- Use named volumes:

```
docker volume create my_volume
```

```
docker run -v my_volume:/container/data <image_name>
```

- For Docker Compose:

```
volumes:
  app_data:
services:
  app:
    volumes:
      - app_data:/container/data
```

Advanced Configuration and Multi-Container Management

11. How do you connect a Docker container to multiple networks?

Answer:

1. Create the networks:

```
docker network create network1
```

```
docker network create network2
```

2. Run the container and attach it to one network:

```
docker run --network network1 --name my-container -d alpine
```

3. Connect the container to the second network:

```
docker network connect network2 my-container
```

4. Verify connections:

```
docker network inspect network1
```

```
docker network inspect network2
```

12. How do you set resource limits for a container?

Answer:

- Use the --memory and --cpu options when running a container:

```
docker run --memory=512m --cpus=1 <image_name>
```

- Example to limit:
  - **Memory** to 512MB.
  - **CPU** usage to 1 core.
- For docker-compose:

```
version: '3.7'
```

```
services:
```

```
 app:
```

```
  image: my-app
  deploy:
    resources:
      limits:
        memory: 512M
        cpus: "1.0"
```

13. How do you share environment variables across multiple containers?

Answer:

1. Use an .env file:
    - Example .env file:

```
DB_USER=root
DB_PASSWORD=secret
```

    - Reference in docker-compose.yml:

```
version: '3.7'
services:
  app:
    image: my-app
    env_file:
      - .env
```

2. Use a shared configuration management tool like HashiCorp Vault or AWS Secrets Manager for production environments.

14. How do you run multiple versions of the same application in Docker?

Answer:

1. Use separate tags for each version:

```
docker run --name app-v1 my-app:v1
```

```
docker run --name app-v2 my-app:v2
```

2. Use different ports for each version:

```
docker run -p 8080:80 my-app:v1
docker run -p 8081:80 my-app:v2
```

3. Alternatively, use Docker Compose with multiple services:

```
services:
  app_v1:
    image: my-app:v1
    ports:
      - "8080:80"
  app_v2:
    image: my-app:v2
    ports:
      - "8081:80"
```

15. How do you troubleshoot high CPU usage in a Docker container?

Answer:

1. Identify the problematic container:

```
docker stats
```

2. Inspect the running processes:

```
docker exec <container_id> top
```

3. Analyze the logs:

```
docker logs <container_id>
```

4. Profile the application to detect bottlenecks.
5. Use strace or similar tools inside the container:

```
docker exec <container_id> strace -p <process_id>
```

16. How do you build a multi-stage Dockerfile?

Answer:

- Multi-stage builds optimize image size by separating build and runtime stages:

```
# Build stage
FROM node:16 AS builder
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

- Build the image:

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

17. How do you run a container with limited network access?

Answer:

- Use a custom Docker network with restricted rules:

```
docker network create --subnet=192.168.1.0/24 my-network
docker run --network my-network <image_name>
```

- Use firewall rules on the host to restrict outgoing traffic.
- Use --network none to completely isolate the container from the network:

```
docker run --network none <image_name>
```

18. How do you roll back to a previous version of an image?

Answer:

1. Pull the previous version:

```
docker pull my-app:v1
```

2. Stop the current container:

```
docker stop <container_id>
```

3. Start a new container with the older version:

```
docker run --name app-v1 -d my-app:v1
```

19. How do you migrate data from one volume to another?

Answer:

1. Create a temporary container to copy data:

```
docker run --rm -v old_volume:/from -v new_volume:/to busybox sh -c "cd /from && cp -r . /to"
```

2. Verify data in the new volume:

```
docker run --rm -v new_volume:/data busybox ls /data
```

20. How do you monitor logs across multiple containers?

Answer:

1. Use the docker logs command for individual containers:

```
docker logs -f <container_id>
```

2. Use a centralized logging system like ELK stack or Fluentd.
3. Use Docker Compose with logging drivers:

```
version: '3.7'
services:
```

```
app:
  image: my-app
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"
```

Network, Security, and Resource Management

21. How do you handle a Docker image that becomes outdated or bloated?

Answer:

1. Regularly clean up unused images:

```
docker image prune -a
```

2. Use a clean base image and build minimal layers:
    - Start with a minimal base image like alpine.
    - Avoid installing unnecessary packages.
3. Remove intermediate files in the Dockerfile:

```
RUN apt-get update && apt-get install -y curl && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*
```

4. Automate rebuilding images with CI/CD pipelines to keep dependencies up to date.

22. How do you troubleshoot a container that is unreachable on its published ports?

Answer:

1. Verify container status:

```
docker ps
```

2. Check port mappings:

```
docker inspect <container_id> | grep "Ports"
```

3. Ensure the host port is not already in use:

```
netstat -tuln | grep <port>
```

4. Verify firewall rules and network policies on the host.
5. Test connectivity using curl or telnet from the host to the container's published port.

23. How do you configure logging for a Docker container?

Answer:

1. Use logging options when running a container:

```
docker run --log-driver json-file --log-opt max-size=10m --log-opt max-file=3 <image_name>
```

2. Configure logging in docker-compose.yml:

```
logging:
  driver: "json-file"
  options:
    max-size: "10m"
    max-file: "3"
```

3. Integrate with centralized logging systems like Fluentd or ELK by specifying a logging driver.

24. How do you restart a container automatically with a delay between retries?

Answer:

- Use --restart policies with Docker Compose:

```
version: '3.7'
services:
  app:
    image: my-app
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
```

```
    max_attempts: 3
```

- For Docker CLI:

```
docker run --restart on-failure:<max_attempts> <image_name>
```

25. How do you prevent data loss when recreating a container?

Answer:

1. Use named volumes:

```
docker volume create my_data
```

```
docker run -v my_data:/app/data <image_name>
```

2. Backup the volume before recreating the container:

```
docker run --rm -v my_data:/data -v $(pwd):/backup busybox tar czf /backup/data.tar.gz /data
```

3. Store configurations in environment variables or configuration files mounted as volumes.

26. How do you manage multiple versions of Docker Compose files for different environments?

Answer:

1. Create separate files for each environment:
    - docker-compose.dev.yml
    - docker-compose.prod.yml
2. Override the base configuration using -f:

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up
```

3. Use COMPOSE_FILE environment variable:

```
export COMPOSE_FILE=docker-compose.prod.yml
```

```
docker-compose up
```

27. How do you scale a service in Docker Compose?

Answer:

1. Use the --scale option with docker-compose:

```
docker-compose up --scale app=5
```

2. Define scaling in the Compose file:

```
version: '3.7'
services:
  app:
    image: my-app
    deploy:
      replicas: 5
```

3. Ensure the application can handle multiple replicas (e.g., stateless design, shared storage).

28. How do you expose a Docker service to the internet securely?

Answer:

1. Use a reverse proxy like NGINX:

```
docker run -d -p 80:80 -p 443:443 nginx
```

2. Secure traffic with SSL/TLS using Let's Encrypt or self-signed certificates.
3. Limit exposed ports:

```
docker run -p 8080:8080 --expose 8080 <image_name>
```

4. Implement firewall rules to restrict access.
5. Use a Docker network to separate internal and external traffic.

29. How do you inspect the filesystem of a running container?

Answer:

1. Use docker exec to start a shell in the container:

```
docker exec -it <container_id> /bin/bash
```

2. Explore the filesystem directly:

```
ls -al /path/to/directory
```

3. Mount the container's filesystem to the host:

```
docker cp <container_id>:/path/to/file /host/path
```

30. How do you clean up unused Docker objects (images, containers, volumes, networks)?

Answer:

1. Remove stopped containers:

```
docker container prune
```

2. Remove dangling images:

```
docker image prune
```

3. Remove unused volumes:

```
docker volume prune
```

4. Clean up all unused objects:

```
docker system prune -a --volumes
```

Operational Challenges and Optimization

31. How do you configure a container to use a specific DNS server?

Answer:

1. Use the --dns option when running a container:

```
docker run --dns=8.8.8.8 <image_name>
```

2. Configure DNS servers globally in /etc/docker/daemon.json:

```
{
  "dns": ["8.8.8.8", "8.8.4.4"]
}
```

Restart the Docker service to apply changes:

```
sudo systemctl restart docker
```

3. Verify DNS configuration in the container:

```
docker exec <container_id> cat /etc/resolv.conf
```

32. How do you restrict access to a container running on a specific IP address?

Answer:

1. Bind the container to a specific IP on the host:

```
docker run -p 192.168.1.100:8080:80 <image_name>
```

2. Use an overlay network and limit container-to-container communication:

```
docker network create --subnet=192.168.1.0/24 my-network
docker run --network=my-network <image_name>
```

3. Apply firewall rules to restrict access to specific IPs.

33. How do you handle a situation where a container is consuming too much memory?

Answer:

1. Set memory limits during container creation:

```
docker run --memory=512m <image_name>
```

2. Monitor container memory usage with docker stats:

```
docker stats <container_id>
```

3. Check logs for memory-related errors:

```
docker logs <container_id>
```

4. Optimize the application code and reduce memory-intensive operations.

34. How do you run a container in detached mode and attach to it later?

Answer:

1. Start the container in detached mode:

```
docker run -d <image_name>
```

2. Attach to the container:

```
docker attach <container_id>
```

3. Alternatively, use docker exec to open a new session:

```
docker exec -it <container_id> /bin/bash
```

35. How do you limit the number of processes inside a container?

Answer:

1. Use the --pids-limit option:

```
docker run --pids-limit=100 <image_name>
```

2. Configure this limit in docker-compose.yml:

```
version: '3.7'
services:
  app:
    image: my-app
    deploy:
      resources:
        limits:
          pids: 100
```

3. Verify the limit within the container:

```
cat /proc/sys/kernel/pid_max
```

36. How do you restart a container with updated environment variables without stopping the current one?

Answer:

1. Use docker update to modify the environment variables:

```
docker update --env NEW_VAR=value <container_id>
```

(Note: This works only for certain updates; restarting is generally required for env changes.)

2. Stop and start the container with new variables:

```
docker stop <container_id>
```

```
docker run -e NEW_VAR=value <image_name>
```

3. Use docker-compose to apply changes:

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

37. How do you ensure a Docker container always starts on system boot?

Answer:

1. Use the --restart always policy:

```
docker run --restart always <image_name>
```

2. For docker-compose, define the restart policy:

```
version: '3.7'
services:
  app:
    image: my-app
    restart: always
```

3. Enable Docker to start on boot:

```
sudo systemctl enable docker
```

38. How do you secure a Docker container running a sensitive application?

Answer:

1. Use non-root users:

```
RUN adduser --disabled-password appuser
USER appuser
```

2. Limit container capabilities:

```
docker run --cap-drop=ALL <image_name>
```

3. Enable read-only filesystem:

```
docker run --read-only <image_name>
```

4. Use secrets management tools for sensitive data.
5. Scan the image for vulnerabilities using tools like Trivy.

39. How do you connect a container to a running database container?

Answer:

1. Use a shared Docker network:

```
docker network create my-network
docker run --network=my-network --name=db postgres
docker run --network=my-network my-app
```

2. Use the container name as the hostname:
    - Example configuration in the app:

```
DATABASE_HOST=db
DATABASE_PORT=5432
```

3. Test connectivity from the application container:

```
docker exec <app_container_id> ping db
```

Advanced Scenarios, Security, and Deployment

40. How do you handle image versioning for a CI/CD pipeline?

Answer:

1. Tag images with unique identifiers:

```
docker build -t my-app:1.0 .
```

2. Use Git commit hashes or build numbers for tagging:

```
docker build -t my-app:$(git rev-parse --short HEAD) .
```

3. Automate the tagging and push in CI/CD pipelines:

```
docker build -t my-app:${CI_COMMIT_TAG} .
docker push my-app:${CI_COMMIT_TAG}
```

4. Use a latest tag for the most recent stable version.

41. How do you run a container in privileged mode, and why would you need it?

Answer:

- Privileged mode grants the container access to host resources, similar to root access.
- Run a container in privileged mode:

```
docker run --privileged <image_name>
```

- Common use cases:
  - Running Docker inside Docker (dind).
  - Accessing hardware devices like USB or GPU.
- **Caution**: Avoid using privileged mode in production due to security risks.

42. How do you manage multiple containers with different configurations in Docker Compose?

Answer:

1. Define separate services in the docker-compose.yml file:

```
version: '3.7'
services:
  app:
    image: my-app
    ports:
      - "8080:8080"
    environment:
      - ENV=production
  db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=secret
```

2. Use multiple Compose files for overrides:

```
docker-compose -f docker-compose.yml -f docker-compose.override.yml up
```

3. Pass environment-specific configurations using .env files.

43. How do you debug slow container startup times?

Answer:

1. Analyze logs:

```
docker logs <container_id>
```

2. Profile the application startup process inside the container:

```
docker exec -it <container_id> /bin/bash
```

3. Check health checks in the Compose file or Kubernetes deployment.
4. Investigate dependencies like database or API latency.

44. How do you clean up exited containers automatically?

Answer:

- Use the --rm option when starting a container:

```
docker run --rm <image_name>
```

- Remove all exited containers:

```
docker container prune
```

- Automate cleanup with a cron job or CI/CD pipeline:

```
docker rm $(docker ps -a -q -f status=exited)
```

45. How do you implement health checks in a Docker container?

Answer:

- Define health checks in the Dockerfile:

```
HEALTHCHECK --interval=30s --timeout=5s --retries=3 CMD curl -f http://localhost:8080/health || exit 1
```

- Use docker inspect to monitor health status:

```
docker inspect <container_id> | grep Health
```

- For Docker Compose:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
  interval: 30s
  timeout: 5s
  retries: 3
```

46. How do you back up and restore a Docker volume?

Answer:

1. Backup:

```
docker run --rm -v my_volume:/data -v $(pwd):/backup busybox tar czf /backup/backup.tar.gz /data
```

2. Restore:

```
docker run --rm -v my_volume:/data -v $(pwd):/backup busybox tar xzf /backup/backup.tar.gz -C /data
```

3. Automate backups using a scheduled job or script.

47. How do you deploy a multi-tier application using Docker Compose?

Answer:

- Example docker-compose.yml for a multi-tier app:

```
version: '3.7'
services:
  frontend:
    image: frontend-app
    ports:
      - "3000:3000"
    depends_on:
      - backend
  backend:
    image: backend-app
    ports:
      - "5000:5000"
    depends_on:
      - db
  db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=secret
```

- Start the application:

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

48. How do you handle secrets securely in Docker Compose?

Answer:

1. Use the secrets feature in Compose version 3.1+:

```
version: '3.7'
services:
  app:
    image: my-app
    secrets:
      - db_password
secrets:
  db_password:
    file: ./db_password.txt
```

2. Ensure the secrets file is excluded from version control (.gitignore).
3. Use external tools like HashiCorp Vault or AWS Secrets Manager for production.

49. How do you run a container with GPU support?

Answer:

1. Install NVIDIA Container Toolkit:

```
sudo apt-get install -y nvidia-container-toolkit
```

2. Run the container with GPU support:

```
docker run --gpus all nvidia/cuda:11.0-base nvidia-smi
```

3. Specify GPU constraints:

```
docker run --gpus '"device=0,1"' <image_name>
```

50. How do you secure Docker images before deployment?

Answer:

1. Use tools to scan for vulnerabilities:

```
trivy image <image_name>
```

2. Minimize the image size using smaller base images like alpine.
3. Avoid embedding secrets in the image.
4. Implement multi-stage builds to exclude unnecessary files:

```
FROM builder AS build-stage
# Build application

FROM alpine AS runtime
# Copy only necessary artifacts
COPY --from=build-stage /app /app
```

5. Sign images using Docker Content Trust (DCT):

```
export DOCKER_CONTENT_TRUST=1
```

Conclusion

Docker has revolutionized the way applications are built, deployed, and managed, offering unparalleled flexibility and consistency in software environments. However, effectively leveraging Docker requires not only a solid understanding of its core concepts but also the ability to address real-world challenges that arise during development, deployment, and maintenance.

This collection of 50 scenario-based questions and answers provides a comprehensive guide to navigating these challenges. From debugging failing containers to implementing advanced security measures and scaling multi-tier applications, each scenario offers practical insights and actionable solutions. These scenarios are not just theoretical; they reflect common issues faced by developers, DevOps engineers, and system administrators in real-world environments.

By mastering these scenarios, you can:

- Build a deeper understanding of Docker's features and best practices.
- Enhance the efficiency and security of your containerized applications.
- Solve complex issues confidently and optimize workflows for better performance.

Docker continues to play a pivotal role in modern software engineering, and staying prepared for its challenges is key to maximizing its potential. Use this guide as a reference to refine your skills and ensure your Docker deployments are reliable, secure, and production-ready. With the knowledge gained, you are better equipped to tackle any Docker-related scenario, making you a valuable asset in any containerized application development or DevOps environment.